1995

Investigation of the use of the object-oriented paradigm in the construction of a triple store based on dynamic hashing.

Rajesh. Shenoy
University of Windsor

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

INVESTIGATION OF THE USE OF THE OBJECT-ORIENTED PARADIGM IN THE CONSTRUCTION OF A TRIPLE STORE BASED ON DYNAMIC HASHING

by

Rajesh Shenoy

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree of
Master of Science at the
University of Windsor

Windsor, Ontario, Canada
1994

ISBN 0-612-01486-X

Name RAJESH SHENOY

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE.
SUBJECT TERM

0984
SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

Abstract

Any set of knowledge, no matter how complex, can be represented as a collection of triples, each of which consists of a subject identifier, relationship name, and object identifier. In order to support applications in natural language processing, it is advantageous if a triple can be retrieved using any field or a combination of fields as key. An appropriate data structure to support this type of access consists of multiple copies of the collection of triples, each of which is structured as a dynamic hash table. Such a data structure has a number of features which could cause problems in implementation. In particular, the multiple copies of the data could lead to inconsistencies on update. Also there is scope for code re-use owing to the fact that many of the operations on the multiple copies of the data are similar. The thesis investigated in this dissertation is that the object-oriented paradigm is well suited to the construction of a triple store based on dynamic hashing.

The object-oriented paradigm provides a way to structure and manage complex relationships among a large number of system components. The object-oriented features of class, inheritance, and encapsulation facilitates construction, debugging, and maintenance of code in the triple-store system where there is a potential for reuse.

In order to determine if these features are advantageous in this application, a triple-store was designed and implemented in the object-oriented language Eiffel and the design and code were analyzed and conclusions drawn.

To my parents and my brother

Acknowledgments

First and foremost, I would like to thank my supervisor, Dr. Richard Frost. Dr. Frost is a gem of a person and I will ever remain indebted to him for his invaluable help, guidance, and advice to successfully complete my Masters degree and to establish myself in Canada. Without his active and patient supervision, it would have been impossible to complete my thesis within this time frame. I would also like to thank Dr. Subir Bandyopadhyay for his active support throughout my Master's program. Dr. Liwu Li's course on Advanced Database Management Systems helped me to get a grasp of this area. I am thankful to him for his valuable tips and suggestions throughout my Master's program. I am also grateful to Dr. Joan Morrissey and Dr. Robert Pinto for their valuable suggestions and comments on this thesis. I thank the wonderful secretaries of the School of Computer Science Mary, Margaret, and Josette for their help. I wish to thank all my colleagues in the *grad. lab.* for their friendship and support. Finally, I would like to thank Mr. Walid Mnaymneh for all help throughout the various stages of this work.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1 INTRODUCTION

§ 1.1 Background and thesis statement

Database-management systems have been widely used in data processing environments since their introduction in the late 1960's. The success of database management systems over the traditional file-management systems is due to their inherent support of data sharing, independence, consistency, and integrity. To provide a formal framework for the representation and manipulation of data, a database system is usually organized according to a data model. The binary-relational view of the universe simplifies the database-system design phase. Systems designed using the binary-relational view may be implemented using a binary-relational storage structure. Use of this structure results in a number of advantages such as simplified system design and improved data independence. However, very few systems have been built using these structures, the main reason perhaps being the difficulty to implement these structures efficiently.

Representations of binary-relationships which are in the form of triples can be inserted, deleted, or retrieved from a binary-relational storage structure. Two different hash functions using a dynamic hashing scheme act on the triple before being inserted into the database. The triple is inserted into seven different tables one for each combination of the keys and so there are seven different representations of the data. As there exists scope for much redundancy, a mechanism must exist which ensures consistency of the data and code.

Also, during the past several years, application of object-oriented concepts has become an important topic of research in various areas of computer science, such as databases, programming languages, knowledge representation, and even computer architecture. The underlying object-oriented concepts are the common threads linking

these topics. However, there is little reported analysis of the use of the object-oriented paradigm in the construction of a systems software or file management. There is little or no known use of the use of pure object-oriented paradigm to construct a basic storage structure of database systems. This study sheds some light on the use of the object-oriented paradigm for such applications.

1.1.1 The thesis statement

The thesis statement to be defended in this report is that:

> The object-oriented paradigm is well suited for the construction of a triple store based on dynamic hashing.

§ 1.2 Importance of the thesis

The binary-relational storage structure based on dynamic hashing, which we shall refer to as a "triple store" from now on, has a number of implementation problems which are listed below:

- The triple store is a two level structure. An index is maintained in a mainstore. This index is basically a hash table and it uses dynamic-hashing techniques. The blocks of data are stored in a diskstore. The mainstore has a pointer to a block on the secondary storage. This structure needs a programming paradigm that supports the construction of programs to manipulate internal and external data structures.
- Dynamic hashing is complex. The triples have to be inserted, deleted, and retrieved from the database. The tree structure on the mainstore keeps on growing as the database size increases. Also, there must be checks for free blocks whenever the triples are deleted from the database. The tree structure would change accordingly

as one level is reduced. There exists a need for a language which supports coding of complex algorithms for dynamic structures.

- As there are multiple hash tables, there is a need for some mechanism to ensure consistency of data.
- Each and every triple has seven different combinations as keys during insertion, deletion, or retrieval from the database. These multiple hash tables use similar hash functions but different keys which include single field keys, two field keys, and three field keys. There is a potential structure which has a need for reuse.
- Each and every triple has to be stored in seven different tables one for each combination of the keys which is used as an identifier. So, there are seven versions of code for insertion, deletion, and retrieval of triples and these seven different versions of code could lead to inconsistency of code. There is a strong need to ensure consistency.

There is a need to implement this structure in a paradigm that takes care of inconsistency and facilitates code reuse. The object-oriented paradigm has shown to be useful in constructing various systems by reusing the code through the inheritance mechanism. It also supports features such as class and generic types which help in maintaining the consistency of the code.

§ 1.3 Work undertaken to support the thesis

The work undertaken that relates to the thesis includes the following:

A literature survey on *Object-Oriented Database-Management Systems*. It was observed that most of the object-oriented database systems were built either on relational database systems or by using the feature of persistence in object-oriented programming languages.

The author familiarized himself with the object-oriented paradigm. He developed skills in the object-oriented languages : C++, GNU Smalltalk, and Eiffel.

The author built the triple-store system using the pure object-oriented language Eiffel and C.

The use of the object-oriented paradigm in this application was analyzed.

§ 1.4 Summary of the conclusions

A summary of the conclusions that can be drawn:

- The analysis of the code for the triple-store supports the claim that the object-oriented paradigm facilitates software structuring for design and reuse.

  Although the code for the triple-store was not reused in another application, some of the code was easily reused within the application. Because the triple store requires seven different versions of the data, the object-oriented paradigm is definitely well-suited to this aspect of the application as explained in section 6.1, section 6.2, and section 6.3.
- Although the triple-store code was not extended in this investigation, an analysis of what would be required if the code were to be modified to ensure minimal commit window (and therefore improve consistency) was carried out. This analysis shows how the object-oriented paradigm would facilitate such a modification of the code as explained in section 6.5.
- The object-oriented code structure facilitates construction, debugging and maintenance as object-oriented programming provides a way to structure and manage complex relationships among a large number of system components. Forcing objects to communicate through a narrow public interface makes it easier to isolate bugs and determine which elements are responsible for the bugs that do occur.

- The object-oriented paradigm facilitates the maintenance of database consistency by providing an 'object-oriented' solution to the problem of minimizing the update commit window. The solution does not cause any structural changes to the system and does not affect the maintainability of the code.
- The message-passing mechanism facilitates modularity as message passing reduces the number of connections among system components and reduces the number of connections, which increases the modularity of system components.
- Eiffel has built-in functions which facilitate the storage of complete mainstore objects structures in a file in binary format which can be retrieved with ease. This simplifies the task of dumping and retrieving the tree structure owing to the fact that there is no need to write special-purpose scanning programs. Eiffel's built-in functions are consistent with the object-oriented paradigm.
- Current implementation of Eiffel did not have a very flexible support for manipulating disk blocks on secondary storage. However, this can be improved with the new release of Eiffel.
- The interface with C gave a procedural look to the system, thereby allowing the programmer to slightly drift from the object-oriented principles. This can cause a serious effect on the maintainability of the system.
- Analysis of the code reveals that an increased use of the interface to C could increase the complexity of the system, thereby raising concern about the object-oriented nature of the application.
- Correctness of the implementation cannot be proved or determined with certainty as the object-oriented paradigm lacks formal semantics.

Chapter 2 BINARY-RELATIONAL STORAGE STRUCTURES

§ 2.1 The binary-relational view

The binary-relational view of the universe is increasingly being used during the data-analysis stage of a database system design [37] [41]. Any set of knowledge, no matter how complex, can be represented as a set of binary relationships. These binary relationships are basically represented as a collection of triples each of which consists of a subject identifier, relationship name, and object identifier. The binary-relational view regards the universe as consisting of entities with binary relationships between them. A binary relationship is an association between the two entities, the subject and the object. A relationship is described by identifying the subject, the type of relationship, and the object.

For example :

IBM	employs	John

Table 1 A triple

N-ary relationships can be reduced to a set of binary relationships by the explicit naming of the implied entity. For example: "John is employed by IBM since January 1, 1993". This can be reduced to a set of triples as follows:

Employment #1	employee	John
Employment #1	employer	IBM
Employment #1	start_date	January 1,1993

Table 2 A set of triples

The binary-relational view can be used in various applications such as natural language processing [16], artificial intelligence [29] as well as database work [40] [11]. Some of the properties of the binary-relational view can be summarized as follows :

- There is no distinction between entities.
- Any set of knowledge or N-ary relationships can be broken down to a set of binary relationships.
- Real, as well as imaginary relationships, can be depicted.

  Consider the following example: "John said Paul thinks that the moon is made of cheese". This can be represented in a binary-relational storage structure as follows:

moon	madeof	cheese	#1
Paul	thinks	#1	#2
John	said	#2	

Table 3 An imaginary relationship represented as triples

§ 2.2 What is a binary-relational storage structure ?

According to Mariani [27] "The binary-relational model first came into prominence in 1974 [4] and was further developed by Senko's work on DIAM (data-independent access model) I and II [38]". All things in the universe can be considered as entities and these entities have a set of binary relationships amongst them. The binary relationships are regarded as belonging to sets called binary relations. These relationships would be between a subject identifier and an object identifier. A structure which can store such binary relationships is known as a binary-relational storage structure.

Consider the following example:

IBM	John
IBM	Peter
IBM	Sam

Table 4 Relation: Employs (Continued) . . .

BNR	Joe
Microsoft	Don

Table 4 Relation: Employs

The above table shows the binary relationship "employs" between the "employer" which is the subject and the "employee", the object.

John	25
Peter	26
Sam	37
Joe	32
Don	23

Table 5 Relation: Age

Similarly, the above table shows the binary-relationship "age" between the "person" and the actual figure which depicts the person's age.

A binary-relational storage structure can be described as a data structure in which representations of binary-relationships can be stored, deleted, and retrieved. These relationships may be stored as triples i.e. subject identifier, relationship, and the object identifier. For example :

IBM	employs	John
IBM	employs	Peter
John	age	25
Peter	age	24

§ 2.3 Advantages/disadvantages of binary-relational storage structures

Several advantages and disadvantages of the binary-relational storage structure have been identified by Frost [16], some of which are listed below. The advantages are:

- It has a simple interface with other modules of the database system. Its interface includes:

  * Insert (This module inserts a triple into the database)
  * Delete (This module deletes a triple from a database)
  * Retrieve (This module retrieves a triple or a set of triples from the database)

  These features are basic and are simple to interface with other systems.

- Data independence is improved. Introduction of a new application program, or modification of an existing one, has minimal impact on the storage structure and consequently has minimal impact on other programs using the storage structure. In general, all that is needed is addition and deletion of some triples.
- File design is no longer necessary. Even if a complete set of requirements is available, and has been analyzed, choosing an appropriate file system design is usually difficult. Also, as the data is broken down and stored as triples, it is not necesary to go through the file design phase as in a conventional database system. If, however, a binary-relational storage structure is available, specification and analysis are not so critical, and problems of file design are reduced. As far as storage and retrieval are concerned, the analyst/designer of the system has to only specify the entity-sets and the binary-relationships that would be represented by the data. This is because all the conceptual access paths are implemented equally and efficiently using the binary-relational storage structure.

- Integration of muti-attribute retrieval requests into the overall system design is facilitated. Multi-attribute queries depend on the overall design of the system. It may be simplified for some queries but there may not exist an efficient access path for some other queries in the design. The situation is simplified if a binary-relational storage structure is used. All the conceptual links between entities can automatically be represented by equally efficient physical access paths. The designer would not have to analyze the query to a large extent as in the case of file system design. So in all, multi-attribute retrieval requests do not affect the choice of the storage structure, and so the task of analyzing, and integrating the results of such analysis with other aspects of the system may be avoided.
- Binary-relational storage structures are particularly apt for the support of an object-oriented database structure [28]. Metadata and the data of a relation can be directly stored in a binary-relational storage structure. Consider the following example: The

PERSON	has_spouse	PERSON
PERSON	has_child	PERSON
PERSON	has_age	age
PERSON	has_sex	sex

data for the metadata specified above is as follows: As seen above, the metadata can

Rick	has_spouse	Jane
Rick	has_child	Robert
Robert	has_age	8
Robert	has_sex	Male

be stored in a similar manner as the data they represent. The object-oriented database can be considered as a high-level interface to the underlying binary-relational storage structure. Data can be inserted, deleted, and retrieved in groups. This approach

provides a dynamic, active aspect to triple stores. The underlying binary-relational model is capable of modelling any set of relationships, no matter how complex.

- The binary-relational model is suitable in supporting the kinds of operations expected in an object-oriented database, i.e. schema evolution [33] [12]. As each and every data is stored as triples, the schema can be broken down into triples and stored in the database. These triples can be very easily inserted, deleted, or retrieved from the database and so it helps in schema evolution.
- The use of dynamic hashing lends efficiency to the retrieval of objects from a database which grows in impredictable ways.

Some of the disadvantages that can identified with the dynamic-hashing triple store described by Frost [16] are as follows:

- It makes inefficient use of the backing store. It stores seven copies of the data and hash table, one for each combination of the key.
- The structure offers more scope for corruption than a conventional multi-attribute retrieval structure.
- Data may be retrieved on a single relationship. Groups of related items may be acquired by issuing several retrieval commands. Consider the following example: Suppose you wish to retrieve the age, designation, and address of a person called Peter Smith. These entities cannot be retrieved as a single group but three different queries would act on the database. These queries include:

  # (Peter Smith, age, ?)

  # (Peter Smith, designation, ?)

  # (Peter Smith, address, ?)

  This query would be answered more efficiently if N-tuples were stored.

- Batch-processing techniques cannot be applied to a binary-relational storage structure to improve performance. In batch processing, all the transactions are collected, sorted and run against a sorted master file to speed up processing. This improvement in the performance cannot be achieved if a binary-relational storage structure was used, as all relevant fields of each master-file record would have to be retrieved independently. Disk access can not be further minimized with batch processing as would have been the case otherwise.

§ 2.4 Use of binary-relational storage structures as base structures for database systems

Many data models have been proposed as conceptual frameworks in the design of database-management systems. However, the binary-relational model, which is based closely on the binary-relational storage structure has received comparatively little attention. Although it is increasingly being used to aid system analysis [16], relatively few database systems have been constructed with this model as a basis. Some of these systems include: the DIAM family [39], the FACT machine [29], NDB [40], and ASDAS [15]. More recently, the functionality of the binary-relational storage structure has been used to support the functionality of an object-oriented database system [27]. The object-oriented database, Oggetto, is considered as a high-level interface to the underlying binary-relational storage structure, and is implemented using the triple store representation of binary-relations. The following illustrates such use of Ogetto system :

Data may be inserted, deleted, or retrieved from this structure. Data may be held as a set of 'triples'. A triple consists of three fields which consists of the subject identifier, relationship name, and the object identifier. Each triple represents a single binary-relationship.

Before the data is inserted into the database, it is reduced to a set of triples. Thus, the input module of the database system would accept the input from the user and reduce it to a set of triples. These triples are validated and if the validation is true, they are inserted into the database. Rules are used to ensure that they do not contradict the schema.

The retrieve module of the database system would translate the specification of the user-specified output to code containing retrieval requests which are issued to the database when the code is executed. There are seven modes of retrieval depending on which triple field, or combination of fields, is used as key. To achieve the necessary speed of retrieval for the seven different modes, data is replicated seven times and held in seven separate hash tables one for each combination of the key.

Insertion of the triple into the database requires the data to be inserted in all seven tables with each combination of the triple as the key. Whereas retrieval of a triple from the database would require only one table to be accessed. For example, retrieve (?, ?, John) would access the table which uses the object identifier as a key and would retrieve all triples whose object identifier has the value 'John'.

An important requirement of a database system is the ability for the schema to evolve. Triple stores are uniquely placed to meet these requirements as the metadata is stored with the data itself, and thus we can apply triple store operations to the metadata.

§ 2.5 Applications of binary-relational storage structures

Binary-relational storage structures are used in a variety of applications. For example:

2.5.1 Use in medical records and data interchange

Medical records [35] are more than a simple log of events. They record the decision making process and support communication amongst those caring for the patient. Different specialities and different users have different requirements, and any standard scheme

for medical data representation or interchange must take this variety of requirements into account. Descriptions based on the binary-relational models offer greater flexibility and fit well into open architectures for document interchange. IRL2, which uses a binary-relational system, is a component of a prototype advanced medical record architecture being developed in conjunction with medical information workstations [35].

2.5.2 Use in a 3D graphical interface

TripleSpace [28] is an experimental system which offers a three-dimensional topology. The possibilities of the binary-relational model for graphical representation is explored. Various data visualization techniques for advanced human-computer interaction is now being developed as a part of the virtual reality movement. Experiments are now being conducted in 3D graphical interfaces to a binary-relational database [28].

2.5.3 Use in the construction of object-oriented database systems

Binary-relational storage structures are particularly apt for the support of an object-oriented database superstructure. Object-oriented database may be considered as a high level interface to an underlying binary-relational storage structure. One such implementation is Oggetto [27], an object-oriented database layered over a triple store. The object-oriented database thus implemented allows experimentation with an object-oriented query language. The functionality of the storage structure is used to support the functionality of the database.

Kopernick [11] is an object-oriented database system, that allows uniform specification of database requests and application programs. The user interface is based on Smalltalk, and the object-oriented data model is represented in terms of classes and messages. This model is implemented on top of a relational database system. The

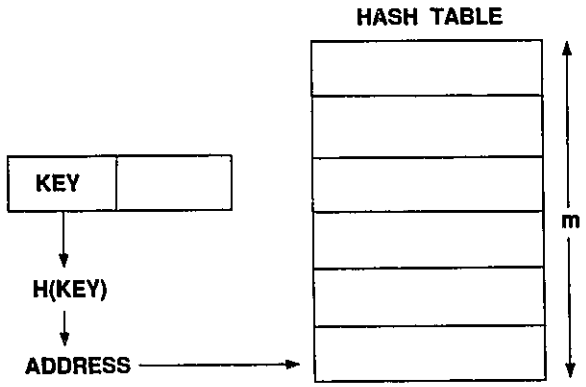
binary-relational view is used as an intermediate level between the underlying database and the conceptual view.

BMRQ [6] is a database interface facility based on graph traversals and extended relationships on groups of entities. A binary-relational model is used as the heart of a coexistent database-system architecture.

Chapter 3 BINARY-RELATIONAL STORAGE STRUCTURES BASED ON DYNAMIC HASHING

§ 3.1 What is dynamic hashing ?

A hash table consists of a number of addressable locations, each of which is capable of holding one or more records. In this structure, the address or location of an identifier, X, is obtained by computing some arithmetic function, *f*, of X. *f*(X) gives the address of X in the table. This address is called as the hash address of X. To place a record in the table, its key is transformed into an address within a range of table addresses by the use of a hash function. Hash tables can provide very fast access to records. There is no need to search through a whole set of records as the key of the record can simply be transformed to an address, using the same hash function that was used to place the record in the table. The data that is stored at this address can then be loaded.



The choice of the hash function should be such that the records are distributed evenly across the table. However, there may be cases when two or more keys get mapped onto the same address. This is called collision. There are different ways to avoid collision as discussed in Frost [17], such as

- Probing or open addressing.
- Collision resolution with chaining.

- Dynamic hash tables.

One of the major disadvantages of the hashing method is the static storage allocation. The number of records must be estimated in advance and physical storage space allocated accordingly. If it is necessary to increase the size of the hash table then all records currently in the table must have their hash codes recalculated and stored at new locations in a larger table.

If storage requirements for a hash table are underestimated, the number of overflow records will be large and this will slow down searching and updating. If the storage requirements are overestimated, storage utilization will be low and storage space wasted. Dynamic hashing was introduced by Larson [25] to avoid this.

In dynamic hashing, the allocated storage space can be easily increased or decreased without rehashing. However, an index, which is structured as a forest of binary trees has to be maintained. The data structure consists of:

- A set of N addressable locations on backing store each of which has the capacity to hold a fixed number of records.
- A set of M cells in mainstore which forms the hash table. This basically forms an index to the blocks. Initially this index constitutes the roots of a forest of empty trees.
- A forest of binary trees is used to accommodate overflow. Each cell is a root of a binary search tree. Initially, no trees exist. However, at some point of time the block would get full when we try to insert a record. When this happens the block is split into two and the index is updated by extending the tree downwards.
- A hash function H1, which is the first hash function, that maps to one of the cells in the hash table. This hash function is a normal hashing function, the only difference is that it identifies a tree and does not directly refer to a block.

The second hash function H2 is applied to the block when it is full and we try to insert a record. This hash function is applied to each and every record in the block. It produces a sequence of binary digits for each record. This binary sequence can be used to determine a unique path in the binary tree, and so insertion, deletion, or retrieval of the record can be carried out very easily. For example a '1' in the sequence indicates one direction (left or right) be taken at a node. A '0' indicates that the other direction be taken.

Ideally, the binary trees that are generated in such schemes should be perfectly balanced. Perfectly balanced trees have a minimal average path length from root to end-node. Record retrieval is fast and storage space is low.

However, as pointed out by Frost [18], random trees are nearly as good as perfectly balanced trees. A random tree is the one constructed from randomly and independently generated binary streams. The function H2 proposed by Frost [18] has two stages:

- The key is converted to an intermediate sequence of binary digits by the direct replacement of characters by the corresponding binary codes in a programmer-defined table [Table 7]. This table is designed such that:
    - * Most frequently used characters are given binary codes with the most even distribution of 0's and 1's.
    - * Characters comprising of frequently used subsets, such as decimal digits and alphabetic characters , are given binary codes such that each subset as a whole has a reasonable even distribution of 0's and 1's.
- The final binary stream is generated by picking bits off the intermediate stream as follows: the first bit of the stream corresponding to the first character is picked off, and then the first bit of the character corresponding to the second character, and so

on. This process is then repeated for the second bit and so on. This process reduces the effect when two or more keys start with the same character.

The corresponding codes for the characters is shown in Table 7. Consider the keys BELL and BELT. The intermediate streams that would be generated are as follows:

BELL	100100	010101	110110	110110
BELT	100100	010101	110110	101001

Table 6 Intermediate Stream generated for id's BELL and BELT

It would take 20 levels to navigate through the above stream to separate the two keys. However, if the bits are picked out in the manner described above, only 8 levels are required.

BELL	10110111
BELT	10110110

Table 7 Final Stream for ids BELL and BELT

0	100011	sp	101010	@	000000	P	101011
1	001011	!	101110	A	010110	Q	011101
2	110010	"	010001	B	100100	R	001110
3	011001	#	000010	C	110101	S	011100
4	100110	£	000100	D	001001	T	101001
5	001101	%	100000	E	010101	U	001010
6	110100	&	101111	F	001100	V	011011
7	010011	'	100001	G	010100	W	010010
8	101100	(	110000	H	111000	X	100010
9	011010	)	001111	I	000111	Y	101101
:	101000	*	010000	J	110111	Z	000110
;	011000	+	011111	K	000101	[	000001
<	111010	'	100111	L	110110	S	111111
=	111011	-	111001	M	110011	]	111100

Table 8 Bit code equivalent of the ASCII character set (Continued) . . .

>	001000	.	010111	N	110001	->	111110
?	011110	/	000011	O	100101	<-	111101

Table 8 Bit code equivalent of the ASCII character set

A triple store is a two level structure: it has an index in the mainstore and a collection of diskblocks, which forms the database, in the diskstore. The index in the mainstore contains a hash table where each cell of the hash table is the root of a binary search tree. The end of the tree contains a pointer to a disk block on the secondary storage. So, at the very start of the database, the structure would look like Fig 1.

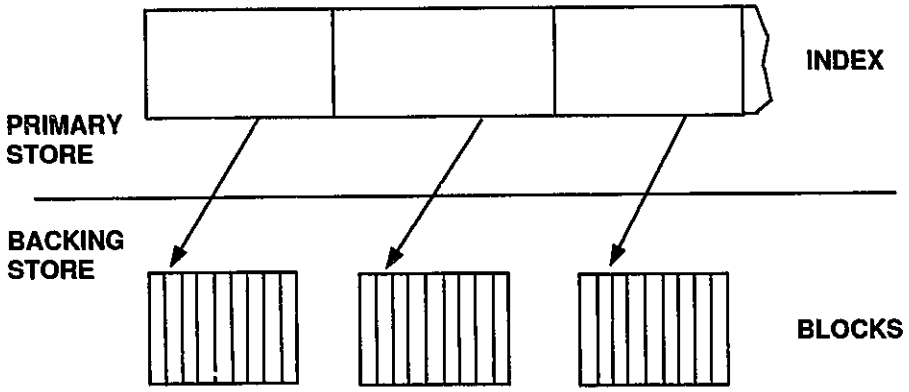


Figure 1 Basic Block

The triple would be inserted into the database based on the identifier which is used as the key. The first hash function is applied to the key to get a position on the hash table. Initially, this would contain a pointer to a disk block on secondary storage and the triple would be inserted in the block. However, as more and more triples get mapped onto the same location on the hash table, a stage would reach when the disk block gets full. The identifier of each and every triple in the full block would then be applied to a second hash function. The second hash function generates a sequence of binary digits. The triples stored in the current block would then be split into two blocks, depending

on whether the next bit in the sequence is a '0' or '1'. Fig 2 shows how the structure would appear when a block is split into two.

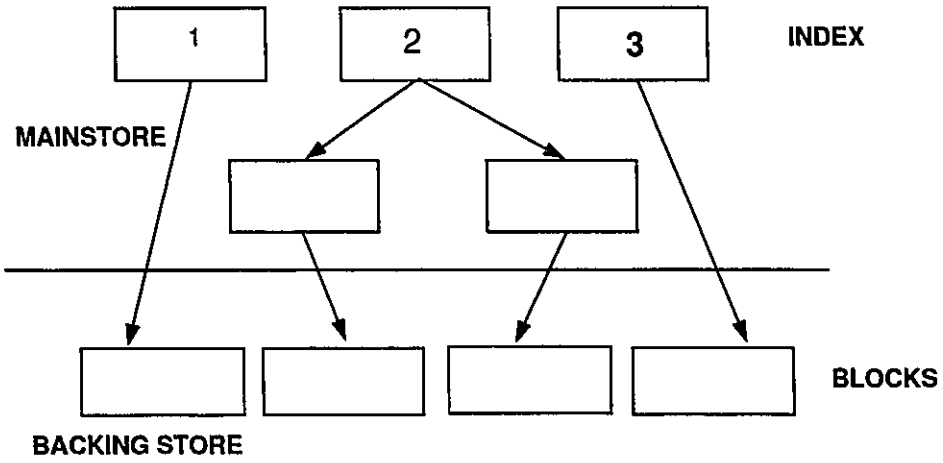


Figure 2 Splitting of the blocks

There may be a case when all the triples would hash onto the same block. Then the second hash function would apply again with the next bit in sequence. This process would go on till either of the blocks is empty. The same hash function at the corresponding bit level would then apply to the triple to be inserted and it would go to the appropriate disk block. When the database has been sufficiently loaded, the index would appear as a forest of binary trees and the structure would appear as shown in figure 3.

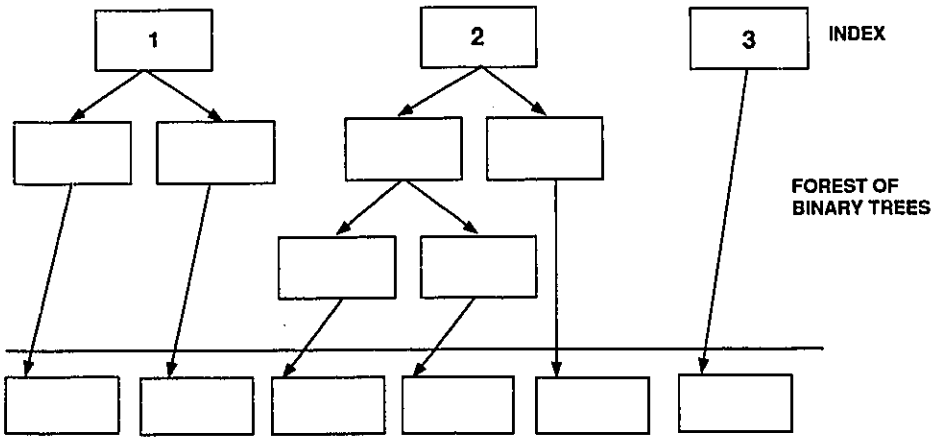


Figure 3 Appearance of the structure when it is sufficiently loaded with data

§ 3.2 Advantages/Disadvantages of dynamic hashing

Dynamic hashing has a number of advantages, some of which are summarized below:

- Groups of records which have the same key are stored together in a single block. If the records exceed the block size then the records are split into two blocks by applying the second hash function H2. Records which have the same key are mapped to the same location and groups of such records are chained together. Groups of records can thus be retrieved using a single disk access. At the most one or two accesses to the backing store is needed to retrieve a record.
- The backing store file is approximately 69% full [17]. Once a block is full, it splits into two blocks each of which is 50% full. These blocks start filling when more triples are being inserted into the database. Empirically, these blocks may either be 50% to 90% full. So in this case, on an average, these blocks may be 75% full. Because the keys of the triples are random, it produces nearly balanced trees on the mainstore. Statistical analysis conducted by Larson [25] show that the blocks would be approximately 69% full. This depends on the two hash functions. The choice of the first hash function would provide an even distribution of the triples across the

hash table and the second hash function provides an even distribution of records in the blocks on secondary storage. So depending on the choice of the functions the blocks would be approx. 69% full.

- It can accommodate collections of data which may vary in size.

Although there are no major disadvantages of dynamic hashing, care must be taken to maintain it; some of which are as follows:

- The use of explicit pointers to represent the tree structure makes for the main store requirements. These pointers to disk blocks must be carefully maintained. Two hash functions act on a triple during insertion, deletion, or retrieval of a triple. The time taken to perform the two hash functions would affect the performance of the triple store.
- Dynamic data structures have to be dumped and recreated. .
- In the preceding section, the records are assumed to be of fixed length. This assumption is not crucial and can be changed to handle variable size blocks on secondary storage by keeping an account of the occupied space by each block. This information must be carefully maintained.

§ 3.3 What is a triple store ?

As explained in section 2.1 and section 2.2, the binary-relational model consists of entities (which are any 'thing' that can be identified and is of interest) which have binary relationships among them. Such a relationship can be represented by a triple of the form

(subject, relation, object).

Consider the following example: "John kicked the ball". In this case, we can say that John is the subject, the ball is the object, and the relationship between the two is that of

the ball being kicked. This is a very simple case as our original fact was a triple. Also, as stated earlier in section 2.1, N-ary relationships can also be represented as a set of triples.

Each triple would have identifiers that are a single field, two fields, or three field keys. The single-field keys would either be the subject, relation, or object. Two-field keys would be a combination of the above. So we would have three, two-field keys. There would be one three-field key. So in all there would be seven keys for each triple. These triples would then be inserted in a table using the identifier as the key. This would result in seven different tables one for each combination of the keys. Triples can be inserted singly and deleted or retrieved in sets. There are seven basic ways in which triples may be retrieved. Using our own notation. the seven basic forms are :

retrieve_table1 (a, ?, ?)

retrieve_table2 (?, b, ?)

retrieve_table3 (?, ?, c)

retrieve_table4 (a, b, ?)

retrieve_table5 (?, b, c)

retrieve_table6 (a, ?, c)

has_table7 (a, b, c)

The seventh table would yield a value true or false depending on whether the triple is in the database. The other tables yield sets of triples which match the given fields. For example, (?, ?, ball) would look into table3 and retrieve a set of triples with 'ball' as the argument in the object field.

§ 3.4 A triple store based on dynamic hashing

The overall appearance of a triple store based on dynamic hashing would be as shown in Fig. 4.

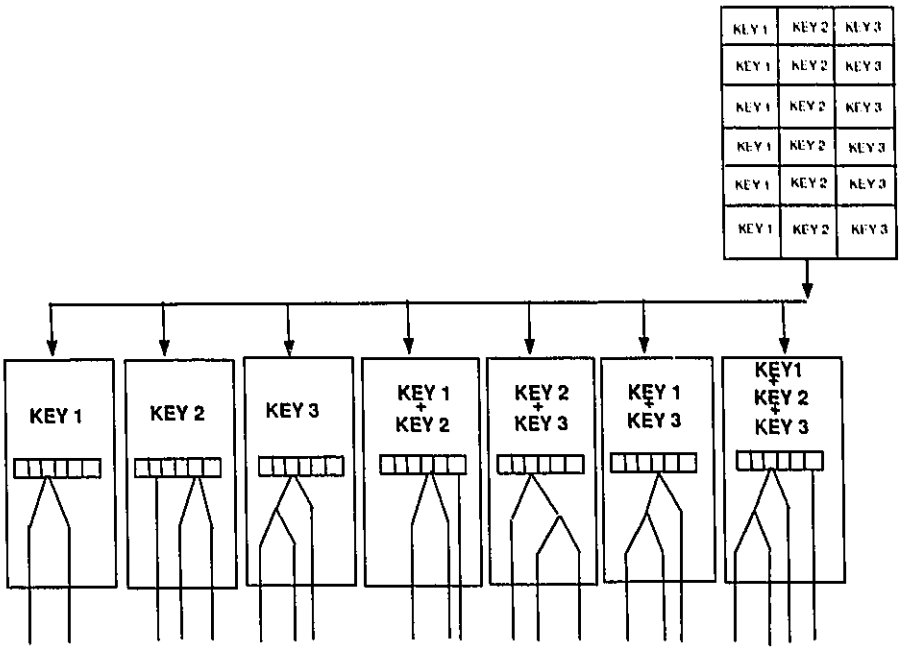


Figure 4 The Overall storage structure

Each and every triple to be inserted into the database would be stored in seven different tables, one for each combination of the key. In conventional hashing, the number of records must be estimated in advance and the physical storage space allocated accordingly. If it is necessary to increase the size of the hash table then all records currently in the table must have their hash codes recalculated and stored at new locations in a larger table. It is difficult to predict the size of all the tables in the triple store using conventional hashing. In dynamic hashing, the allocated storage space could easily be increased or decreased without rehashing, by maintaining a relatively small index, which is structured as a forest of binary trees.

Besides this, as data is stored in seven different tables using dynamic hashing, groups of similar data are chained together and can be retrieved using a single disk access or by following the chain of pointers. At the most one or two accesses to the backing store is needed to retrieve a record.

Chapter 4 THE OBJECT-ORIENTED PROGRAMMING PARADIGM

§ 4.1 Background

4.1.1 The history of object-oriented programming languages

During the past several years, the object-oriented approach to programming and designing complex software systems has received tremendous attention in the programming languages, knowledge representation, and database areas. Although the object-oriented approach has become popular, there has been much confusion and controversy about what "object-oriented" means. The fact that there is no consensus about precisely what objects are is unfortunate in view of the fact that the foundational object-oriented concepts have evolved in three different disciplines: first in programming languages, then in artificial intelligence, and then in databases. Simula-67 [13], a language for computer simulations, is regarded as the first object-oriented programming language. The first appearance of the programming-construct notion *object* was from Simula-67. The discovery was that software objects could be used, not only for simulation, but also for prototyping and application development. Since Simula-67, researchers in programming languages have taken two different paths to promote object-oriented programming, as discussed in [32]. One was the development of new object-oriented languages such as Smalltalk, Eiffel, Trellis/Owl etc. The Smalltalk system is based on the Simula concept of object class. Smalltalk-80 is an interactive pure object-oriented language. In Smalltalk systems, everything is an object. Each class definition is an object. Another was an extension of conventional languages: Flavours, Object Lisp, LOOPS, and Common LOOPS, as extensions of LISP; and so on. Simula-67 motivated an expanded version of C by Stroustrup at the AT&T Laboratory. The new language was called "C with classes" (1980), and

the name was changed to C++ in 1983. In C, once a program exceeds a few thousand lines of code, it becomes so complex that is difficult to grasp as a totality. C++ allows programmers to comprehend and manage larger programs.

The language Eiffel (1988) is a pure object-oriented language that features strong typing to help programmers ensure correctness and robustness of their software. Eiffel has been designed by Bertrand Meyer, one of the founders of the object-oriented paradigm.

Object-oriented programming is a new way to approach the task of programming. Object-oriented programming encourages the programmer to decompose a problem into related subgroups. Each subgroup becomes a self-contained object that contains its own instructions and data that relate to that object. In a way, complexity is reduced and the programmer can manage larger programs. Object-oriented languages have become increasingly popular and can be found in many different varieties.

4.1.2 The characteristics of object-oriented programming languages

There is a considerable controversy about the features that should be called as object-oriented. All object-oriented programming languages share three common features :

4.1.2.1 Encapsulation

Encapsulation is a mechanism that binds together methods and the attributes it manipulates and keeps both safe from outside interference and misuse. In an object-oriented language, attributes and methods may be bound together in such a way that a self-contained *black box* is created. Within the box are all the necessary attributes and methods. When attributes and methods are linked together in this fashion, an object is created and so it supports encapsulation.

Encapsulation associates the behavior functions of an object with its data items. Abstraction is the act of separating the essential qualities of an object from the details of how it works or how it is composed. Abstraction enforces information hiding.

Within an object, attributes, methods, or both may be private to that object or public. Private attributes and messages are known to and accessible only by another part of the object. When an attribute or method is public, other parts of your program may access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private parts of the object. Any modification of the implementation will not affect the application programs so long as the interface of the object has not been changed.

So basically, an object is a variable of a user-defined type. Each time an object is defined, in reality, a new data type is created.

4.1.2.2 Class

A group of similar objects are described by the definition of a class. An object is a logical unit, a class is a program unit.

The features of encapsulation, abstraction, and information hiding are implemented by class.

4.1.2.3 Inheritance

The software elements defined in a class definition can be reused in the definition of another class. The former class is known as a superclass and the latter, a subclass. The subclass inherits properties from the superclass.

Inheritance is the process by which one object can acquire the properties of another. More specifically, an object can acquire a general set of properties to which it can add

those features that are specific only to itself. Inheritance is important because it allows the object to support the concept of hierarchical classification.

The implementation of a data variable or a function in the subclass can override that of the data variable or function in the superclass.

Multiple inheritance is possible in most object-oriented languages. In an object-oriented system, a class may have any number of subclasses. However, some systems allow a class to have any number of superclasses. In the former, a class inherits attributes and methods from only one class; this is called single inheritance. In the latter, a class inherits attributes and methods from more than one superclass; this is called multiple inheritance.

Example: Smalltalk80

```
classname WordLink
superclass Link
instance variable names word
class methods

for: aString
     ^(self new) word: aString
word
     ^word
word: aString
      word <- aString
....
list add: (WordLink for: #one)
```

Example: Eiffel

```
class WordLink inherit
    Link
export
```

```
    getWord, setWord
feature
    word: Real;

    getWord: String is
       return word
    end;

    setWord (s: String) is
       word := s
    end;
    ...
end
```

The above class definition can be used to define a data item in another class.

```
w: WordLink;
....
w.Create;
w.setWord("Hello");
...
```

Example: C++

```
class WordLink: public Link
{
 private:
     char *word;
 public:
     WordLink(void);
     WordLink(char *s) {
        *word = *s;
     }

     char *getWord() {
        return word;
```

```
    }

    void setWord(char *s) {
      *word = *s;
    }
    ...
};
```

The above class definition can be used to define a data item in other functions:

```
WordLink w("Hello");

cout << w.getWord();
```

will print "Hello" on the screen.

4.1.2.4 Polymorphism

Polymorphism is the quality that allows one name to be used for two or more related but technically different purposes. Polymorphism allows one name to be used to specify a general class of actions. Within a general class of actions, the specific action to apply will be determined by the type of data.

More generally, the concept of polymorphism in the object-oriented paradigm is the idea of "one interface, multiple methods" [31]. This means that it is possible to define a generic interface to a group of related activities. However, the specific action depends on the data. The advantage of polymorphism is that it helps to reduce the complexity by allowing the same interface to be used to specify a general class of action.

Polymorphism can be applied to both functions and operators. Virtually all programming languages contain a limited application of polymorphism. For example, the '+' sign is used to add integers, long integers, characters, and floating point values. The key point about polymorphism is that it allows you to handle greater complexity by allowing the creation of standard interfaces to related activities. The execution of a message is

dynamically determined by the types of the arguments and/or the type of the message receiver.

Consider the following example in C++:

```
class Employee {
     float salary;
     virtual void salaryIncrease() = 0;
     ...
};

class Faculty : public Employee {
    ...
    void salaryIncrease() {
        salary *= 1.07;
    }
   ...
};

class Sessional:public Employee {
    ...
    void SalaryIncrease() {
    } //Yes, 0 increase
    ..
};
Employee *e; Faculty *f; Sessional *s; ...
e = f;  e -> salaryIncrease();
e = s;  e -> salaryIncrease();
```

Inheritance is not just a module combination and enrichment mechanism. It also enables the definition of flexible entities that may become attached to objects of various forms at run time. An assignment of the form `a:=b` is permitted not only if *a* and *b* are of the same type, but more generally if *a* and *b* are of reference types A and B based

on classes A and B such that B is a descendant of A.

In the above example *class* Faculty and *class* Sessional are descendants of *class* Employee. Both the subclasses have their own version of the function *salaryIncrease()*. The version to use in any call is determined by the run-time form of the target.

4.1.3 Why is object-oriented programming gaining popularity?

4.1.3.1 Software-engineering crisis

Hardware components can be repeated easily so that one design can be used millions of times. "Why must every new software development start from scratch?"[9]. At the NATO workshop on the software crisis [9](1968), an approach of "mass produced software components" was advocated.

In software development, a number of basic patterns are repeated over and over again. There are some non-technical obstacles for reuse of software components. Object-oriented programming provides one method for supporting the re-use of components. A collection of well-developed classes can be easily integrated into a new system with inheritance and overriding.

4.1.3.2 Reusability and extensibility

It is easy to develop object-oriented systems that can accommodate new classes. For example, if we add a new class Secretary as a subclass of Employee, the expression

```
e -> salaryIncrease();
```

may also be applied to an object *e* of Secretary.

Because of information hiding, object-oriented software maintenance is easier and cheaper. In information hiding, all information about a module should be private to the module unless it is specifically declared public. Application of this principle assumes

that every module is known to the rest of the world (that is to say, to designers of other modules) through an interface. Although there is no absolute rule for deciding what should be the interface and what should be secret, the general idea is clear: the interface should be the description of the module's function or functions; anything that relates to the implementation of these functions should be kept private, so as to preserve other modules from later reversals of implementation decisions. As a result, user modules will not suffer from any change in implementation.

4.1.3.3 Commercial compilers and class libraries

More and more compilers of object-oriented languages are available. C++ compilers are available for almost every platform. Commercial and public domain systems for Smalltalk languages have been available long ago, and Eiffel compilers have been installed in many universities.

4.1.4 Applications of object-oriented programming

Object-oriented languages have been applied to develop various systems, which include compilers, operating systems, knowledge-base, and database systems. Since the notion of objects naturally reflects the structures of hardware components, the approaches of object-oriented programming and object-oriented analysis and design have been extensively applied to CAD/CAM. Some of the applications include:

- Object-oriented analysis and design for CAD/CAM [19].
- Object-oriented databases [27].

4.1.5 Applications of object-oriented database systems

A database-management system may or may not be developed in an object-oriented language, but it must support the core concepts of object-oriented programming to be qualified as an object-oriented database management system.

The data model of an object-oriented database-management system must support the concepts of encapsulation and unique object identification, attributes and methods, class, and inheritance.

One reason for the development of object-oriented database management systems is that the relational and other conventional database systems are no longer suitable to new computer applications like CAD, CAE, CASE, CAM, multimedia systems, which have unstructured data.

The data-definition language of an object-oriented database management systems must support the definition of classes; the data manipulation language must allow a user to create, modify, and retrieve an object, a group of related objects, or the attributes of the objects. The access unit must be an object. Both navigational and declarative access should be possible for an object-oriented database management systems.

There is no ANSI standard for object-oriented database management systems. However, the object data management group (ODMG) participants from workstation and object-oriented database industries have developed ODMG-93 Standard version 1.0. The ODMG represents over 80% of the total marketplace and they proposed the adoption of ODMG-93 as an ANSI/ISO standard [8].

§ 4.2 Eiffel

4.2.1 Introduction

Eiffel is a software-development environment intended for developers who want to achieve the best in software quality: correctness, reusability, extensibility, efficiency, and portability. It is developed by Interactive Software Engineering Inc. It includes a language, a library of reusable software components, and a collection of supporting tools.

The company, Interactive, designed the language and produced the first implementation of Eiffel. To promote the development of a market of reusable Eiffel components, the creators of Eiffel have explicitly stated that the language design is not proprietary, and that any organization is welcome to produce other implementations, tools, or libraries [2].

4.2.2 Overview of the Eiffel language

The aim of Eiffel is to specify, design, implement, and modify quality software. This section gives an overview of the language and a presentation of the semantics used for syntactic and semantic construct descriptions. The architectural notions of the system are explained such as class and feature, the inheritance and client relations between classes, deferred classes, generic classes, assertions, and the feature adaptation mechanism based on inheritance.

4.2.2.1 Class and feature

The constituents of Eiffel software are called classes and have the following properties:

- They are the building blocks of Eiffel programs.
- The classes may be assembled into a executable system and are the basis of the typing system.
- A class, at compile time, contains a collection of features which are to be features in every instance. A feature is a constant, variable, procedure or function.
- There is one class per file and vice versa for the source code.

The following concepts provide the basis for structuring Eiffel software :

- An Eiffel system results from the assembly of one or more classes to produce an executable unit with one of them being the root of the system.

- A cluster is a set of related classes. A universe is a set of clusters, out of which developers will pick classes to build systems.
- An Eiffel system can be assembled using the *es* command on the name of the root class. An executable file is produced if no errors are found.
- Execution of an Eiffel program starts with the creation of one instance of the root class and consists of the execution of the procedure *Create* found in the root class.
- Other objects may be created during execution.

4.2.2.2 Inheritance

Inheritance makes it possible to define a new class by combination and specialization of existing classes rather than from scratch. Class A inheriting from class B means that all the features of class B become features of class A. The options upon inheritance are :

- Renaming of an inherited feature.
- Redefining of an inherited feature.
- Inheritance from more than one class.

The advantage of inheritance is :

- You can reduce the amount of code by reuse.
- You can write a data type as an extension of another data type.
- You can write a data type as an adaptation of another data type.
- You can define data types with variant meaning for the features through dynamic binding.

4.2.2.3 Deferred classes

The inheritance mechanism includes one more major component : deferred routines and classes. A deferred class is a class with at least one deferred feature. A non-deferred

routine or class is said to be effective. A deferred feature is a feature with no default definition for its body. The parameter and return value types are defined. Definition of a deferred feature is eventually expected within a descendent. A deferred feature, inherited by a class not giving it a definition, becomes a deferred feature in the new class.

Dynamic binding provides a high degree of flexibility. If class B conforms to class A (B is A or B is a proper descendant of A) then an object of type B at runtime may occur anyplace where an object of type A is expected. If feature *f* is used on object *a*, and if class B has a different version of *f* than class A, the version in class B is used. This is dynamic binding. Dynamic binding gives the ability to request an operation without explicitly selecting the variant. The choice is made based on the dynamic type of the object.

4.2.2.4 Generic class

A generic class is a definition of a class with one or more formal generic parameters representing arbitrary types. Genericity can be combined with inheritance.

To make a class generic is to give it formal generic parameters representing arbitrary data types. Consider the following examples:

```
ARRAY [G]

LIST [G]

LINKED_LIST [G]
```

These classes describe data structures — arrays, lists without commitment to a specific representation, lists in a linked representation — containing objects of a certain type. The formal generic parameter G represents this type. To derive a directly usable type, a type corresponding to G called an actual generic parameter must be provided. These may either be a basic expanded type such as INTEGER or a reference type. Some of the

possible generic derivations are as follows:

```
il: LIST [INTEGER];

aa: ARRAY[ACCOUNT];

aal: LIST[ARRAY[ACCOUNT]];
```

In the above example, *class* Account is a class that describes bank accounts.

4.2.2.5 Assertions

Eiffel encourages programmers to express formal properties of classes by writing assertions. The following describe each of the assertion clauses each of which are optional :

- The *require* and *ensure* clauses of a routine contain its preconditions and postconditions respectively.
- An invariant of a class indicates properties to be preserved by its exported routines.
- A set of assertions may be placed into a check instruction.
- A loop invariant is a requirement upon loop initialization and after each iteration a loop variant is an integer expression which is to be non-negative and decrease with every loop iteration.

An Eiffel class may be compiled so that all assertions may be monitored, only the preconditions are monitored, or all assertions are ignored.

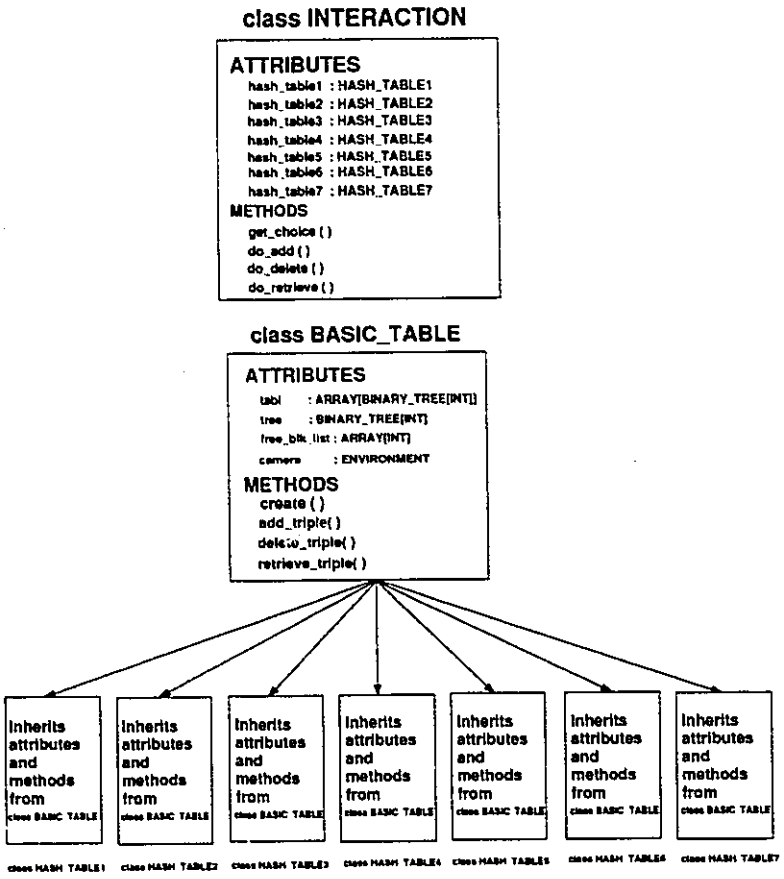
4.2.2.6 Feature adaptation

Multiple inheritance is a frequent occurrence in Eiffel development. For example, most of the classes in the Data Structure Library have two or more parents. There is often a need to adapt the features of parents to a new class. This is achieved through the feature adaptation part of a Parent clause in the class. A feature adaptation part may contain *Rename*, *Redefine*, *New_exports*, *Undefine*, and *Select* subclauses.

Chapter 5 USE OF EIFFEL TO CREATE A BINARY-RELATIONAL STORAGE STRUCTURE

§ 5.1 The design of the overall structure of the programs

The overall design of the system is as follows:



The system consists of two main classes: *class* INTERACTION and *class* BASIC_TABLE. Class BASIC_TABLE has seven subclasses *class* HASH_TABLE1, *class* HASH_TABLE2, *class* HASH_TABLE3, *class* HASH_TABLE4, *class* HASH_TABLE5, *class* HASH_TABLE6, and *class* HASH_TABLE7.

Class INTERACTION interacts with the user. It accepts the user input which may be either to add a triple, delete a triple, or retrieve a set of triples. This class contains

one instance of each table, which is a subclass of *class* BASIC_TABLE, as attributes. It also declares four methods which perform the necessary operations.

getchoice () displays a menu to the user which has four options:

- To add a triple into the database
- To delete a triple from the database
- To retrieve a set of triples from the database.
- To exit the current session

It accepts the user's choice of adding, deleting, and retrieving triples and performs the appropriate action by calling methods *do_add()*, *do_delete()*, or *do_retrieve()*. The method *create()* is the constructor for the *class* INTERACTION.

Create() initializes the hash tables and creates the environment file, *Tree.env*. This environment file stores information about the hash tables and the free-block list. The information about tables is stored in object *tabl* and the free block list is stored in object *free_blk_list* in the environment file. It makes the objects and its dependents persistent. Every object recorded in the environment file has a key associated with it. The constructor *Create()* creates an object *tabl* for each combination of the triple associates it with a key to be stored in the environment file. Similarly, it creates a free-bock list for each table in object *free_blk_list* and stores it with its corresponding key. Finally, it closes the environment file which can be retrieved later.

do_add() accepts from the user the first, second, and third field of the triple which consists of the subject identifier, relationship name, and object identifier. This triple has to be stored in all the seven tables and so each instance of the table performs the message *do_adding()* with the three fields as parameters.

do_delete() accepts the first, second, and third field of the triple as input from the user. As this triple has to be deleted from all the seven tables, each instance of the table invokes the message *do_deleting()* with the three fields as parameters.

do_retrieve() also accepts either the first, second, or third field of the triple as input from the user. It can also accept a combination of the above fields or all the three fields as input. Triples can be retrieved singly or in groups with the combination of the fields as keys. Each instance of the seven tables invokes a message *retrieve_triple()* with the three fields as parameters.

class BASIC_TABLE creates a table with the three basic operations to be performed on the triple. These operations are *initialize()*, *add_triple()*, *delete_triple()*, and *retrieve_triple()*. *add_triple()* in turn calls *record_triple()* which actually performs the insertion of the triple. Similarly, *delete_triple()* calls *del_triple()* and *retrieve_triple()* calls *ret_triple()*, respectively. *record_triple()*, *del_triple()*, and *ret_triple()* are functions in C which are invoked from Eiffel using the Eiffel-C interface. Class BASIC_TABLE can be accessed by other classes only through it's interface. This is specified in the *export* clause while defining the class and consists of methods *initialize()*, *add_triple()*, *delete_triple()*, and *retrieve_triple()*.

initialize() creates the hash tables by assigning the initial block to every cell of the hash table. It has the following parameters:

- *fname* is name of the file which stores the information of the binary hash trees and the free-block list on secondary storage.
- Data is stored in seven different tables one for each combinations of the triple fields as the key. Every table is stored in a different file on this system and the name of this file is passed as a parameter specified by the attribute *tbl_name*.

- Every object recorded in the environment file has a key associated with it. This makes the objects and its dependents persistent. *env_key* and *free_key* are the keys for objects *tabl* and *free_blk_list* respectively which store the information about the hash tables and the free-block list on secondary storage.

The message *initialize()* first opens the environment file, *Tree.env*. This environment file stores information about the hash tables and the free block list. The information about tables is stored in object *tabl* and the free-block list is stored in object *free_blk_list* in the environment file. *initialize()* creates the initial blocks on secondary storage and assigns a block to a cell of the hash table. On initialization of the hash table, a cell is the root of a binary hash tree and so the hash table is a list of binary trees. The message *initialize()* calls a function *assign_intitial_block()* for every cell in the hash table. *assign_initial_block()* returns the address of the newly allocated block which is assigned to each cell. Data is stored in seven different tables one for each combination of the triple fields as the key. The name of the file that stores the block, which is the name of the table, is specified in object *tbl_name*. This object is passed as a parameter by the message *assign_initial_block()*. Finally, the newly created trees for each cell of the table stored in object *tabl* is dumped onto the environment file. This makes the objects and its dependents persistent.

add_triple() first opens the environment file, *Tree.env* which contains information about the hash tables. These hash tables have the root of a binary tree as each element of the list. This information is stored in object *tabl* for each table. It also stores a free-block list in object *free_blk_list*. When allocating disk blocks during addition of a triple, the first block from the object *free_blk_list* is allocated. If this list is empty, then the block pointed to by the free-block pointer is allocated. *add_triple()* has the following parameters:

- *tbl_no* contains the table number at which the triple is to be inserted.
- *flname* is as defined earlier.
- *env_key* and *free_key* is as defined earlier.
- *tbl_name* is as defined earlier.
- *tpl_key* is the key of the triple to be inserted.
- *text1*, *text2*, and *text3* are the three fields of the triple to be inserted which make up the subject identifier, relationship name, and object identifier.

The information about the binary trees and the free blocks are retrieved from the environment file. The first hash function is then applied to the key of the triple to be inserted. This function returns a position in the hash table. The contents at this position is retrieved which may either be a pointer to a disk block on secondary storage or may be a root of a binary tree. If it is a root of the binary tree, then the second hash function is applied to the key of the triple to be inserted. This hash function helps in scanning the tree until it reaches a leaf node where the address of the disk block is stored. The message *record_triple()* is then called which stores the triple at the appropriate location in the disk block. This message returns a string which is then broken up into three elements as follows:

- The first character of the string is stored in the attribute *inserted*. This character determines whether or not, the triple is inserted into the database. If there are free slots in the mapped disk block, the insertion would be successful and *inserted* would return a value 1. However, a condition may arise where all the triples in a full block would be mapped onto the same block at the next level on the tree after applying the second hash function. This would leave no place for the triple to be inserted and so the second hash function would have to applied again on the triple but at a higher

level and *inserted* would return a value 0. This process would go on until *inserted* return 1 which implies that the triple is inserted.

- The second character of the string indicates the number of free blocks used during the insertion of the triple from the freeblock list. The program first checks to see if any blocks are available in this list. If not it allocates the block pointed to by the free-block pointer. The free block list is updated accordingly.
- The remaining characters of the string return the address of the new block, if allocated. The tree level is incremented and the new block address is placed at the suitable location as specified by the second hash function. It returns -1 if no new block is allocated and the tree structure remains intact.

The updated binary hash tree is then stored at its position in the hash table. Finally, the objects *tabl* and *free_blk_list* are updated in the environment file and it is closed.

delete_triple() deletes the triple from the table. It has the following parameters :

- *tbl_no* is as defined earlier.
- *flname* is as defined earlier.
- *env_key* and *free_key* is as defined earlier.
- *tbl_name* is as defined earlier.
- *tpl_key* is as defined earlier.
- *text1*, *text2*, and *text3* is as defined earlier.

delete_triple() first opens the environment file, *Tree.env*. From this environment file, it then retrieves the information about the hash tables and the free-block list and stores it in objects *tabl* and *free_blk_list* respectively. The first hash function is then applied to the key of the triple to be deleted which returns a position on the hash table. This may either point to a disk block on secondary storage or it may be the root of a binary hash

tree. If it points to the root of a binary hash tree, the second hash function is applied to the triple. This helps in scanning the tree until it reaches a leaf node which contains the address of the block which contains the triple. The message *del_triple()* is called which performs the deletion of the triple from the disk block. There may be cases where a block becomes free during deletion. *del_triple()* returns the address of this free block. It returns -1 in other cases. So accordingly, the address of the free block, if any, is inserted into the free-block list and the level of the tree is reduced by 1. The resulting tree structure, after deleting the triple, is then stored at its position in the hash table. Finally, the objects *tabl* and *free_blk_list* are updated in the environment file and it is closed.

retrieve_triple() retrieves the triple either singly or a set of triples from the tables. It has the following parameters:

- *tbl_no* is as defined earlier.
- *flname* is as defined earlier.
- *env_key* and *free_key* is as defined earlier.
- *tbl_name* is as defined earlier.
- *tpl_key* is as defined earlier.
- *text1*, *text2*, and *text3* is as defined earlier.

retrieve_triple() first opens the environment file, *Tree.env*. From this environment file, it then retrieves the information about the hash tables and the free block list and stores it in objects *tabl* and *free_blk_list* respectively. The triples may be retrieved either singly or in sets. The keys of the triples to be retrieved can either be the subject identifier, relationship name, object identifier, or a combination of these. The table to be accessed for the retrieval of the triples would depend upon the pattern in which the retrieval of triples is specified. The fields specified for the retrieval of the triples forms the key to

which the first hash function is applied. This function returns a position on the hash table. This may either point to a disk block on secondary storage or it may be the root of a binary hash tree. If it points to the root of a binary hash tree, the second hash function is applied to the triple. This helps in scanning the tree till it reaches a leaf node which contains the address of the block which contains the triple. The message *ret_triple()* is then called which prints the requested triples.

add_triple() invokes a message *record_triple()* which performs the actual insertion of the triple in the disk block on secondary storage. Similarly, *delete_triple()* invokes the message *del_triple()* and *retrieve_triple()* invokes the message *ret_triple()* which perform the deletion and retrieval of the triples. *record_triple()*, *del_triple()*, and *ret_triple()* are functions in C which are invoked from Eiffel using the Eiffel-C interface.

The structure of a triple in the system is as follows:

```
typedef struct triple {
  char text1[50];
  char text2[50];
  char text3[50];
  int next;
 } TRIPLE;
```

Each field of the triple is a character string of 50 characters in length. The triple structure also contains a field called *next* which stores the address of the next block of a set of triples having the same key. A value of -1 would be stored in this field to indicate the end of the chain. A block is a set of 50 triples and has the following structure:

```
typedef struct block {
  int header;
  TRIPLE record[50];
  int next;
 } BLOCK;
```

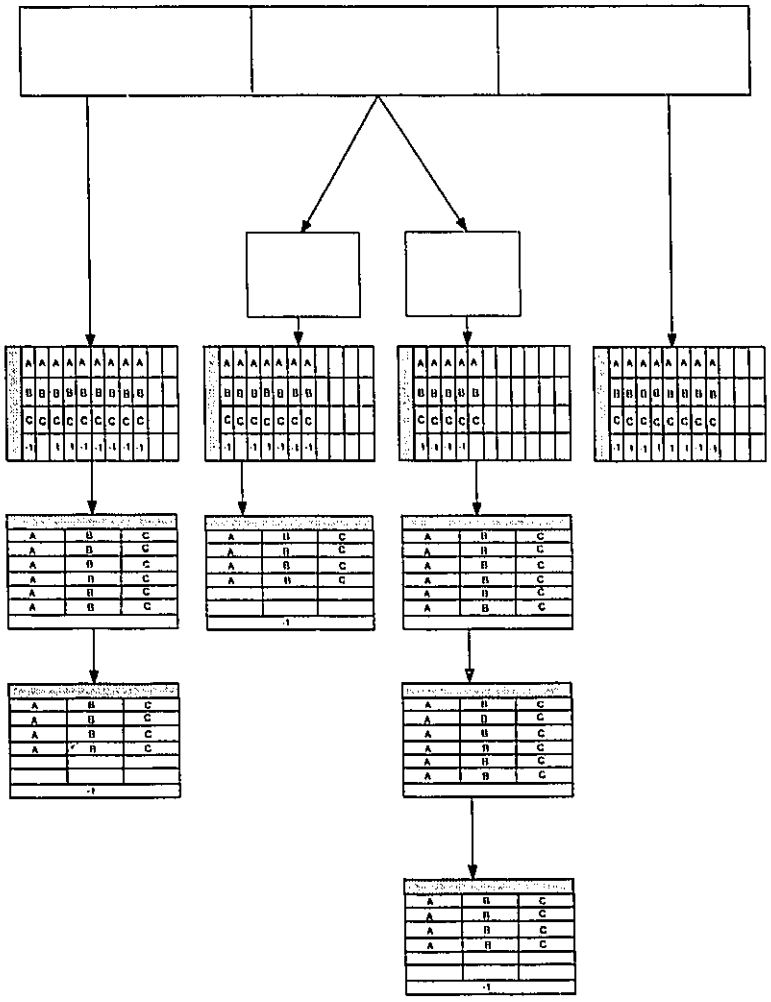
The field *header* indicates the number of records present in the disk block. The field *record* stores the triples as an array of fifty records. Finally, the field *next* stores the address of the next block of a set of triples having the same key.

record_triple() is a C module that performs the insertion of the triple in the disk block. The parameters passed to *record_triple()* are as follows:

- *free_blk_list* is as defined earlier.
- *free_count* is as defined earlier.
- *text1*, *text2*, and *text3* is as defined earlier.
- *tbl_name* is as defined earlier.
- *blk_address* is the address of the block to which the triple is mapped after applying the two hash functions on it's key.
- The address of the block to which the triple is mapped may be stored either at the root of the binary search tree or at the leaf node of tree. The depth of the tree at which this block address is located is stored in attribute *level*. The second hash function uses the depth of the tree to decide the next level of the tree. It is used to reach the leaf node from the root node.
- *tbl_no* is as defined earlier.

The structure of the blocks on secondary storage when it is sufficiently loaded would appear as shown in figure 5:

Figure 5 The Internal Block Structure



The above figure shows the splitting and the chaining of blocks that would occur in the database. Groups of triples map onto blocks after the two hash functions act on its key. When the block gets full, the second hash function would be applied to each and every triple in the full block and it gets spilt into two blocks. This is known as *splitting* of the blocks. However, there may be a situation wherein the incoming triples have the same key. In this case, the triples are chained to each other. Groups of similar

triples are held in a block and groups of these blocks are chained to one another. This is known as *chaining* of the blocks.

For every triple to be inserted, the message *record_triple()* first reads in the main memory, the block pointed to by the *blk_address* parameter. It then checks if a triple with the same key exists in this block.

If true, it goes down to the first free record location on the chain of blocks holding records of the same key. The block header for the corresponding block is incremented by one and the block is stored at its address on secondary storage. If the existing blocks on this chain are full, a new block is allocated. This new block is obtained by first checking the free block list. If the free block list is empty, then the block pointed to by the free block pointer is allocated. The *next* field of the last block on this chain is made to point to the newly allocated block. The triple to be inserted is stored at the first location of the new block. The block header is updated accordingly and the new block is stored thus increasing the chain of similar blocks.

If the block does not contain a triple with the same key as the triple to the inserted, the triple is stored at the next free location of the mapped block. The block header is incremented by one and the block is stored at its address on secondary storage. Now when this block gets full, then the second hash function is applied to each and every triple on the existing block. This function would further spilt the existing block into two. The additional block is obtained in the same manner as described above i.e. the free block list is first checked. If the free block list is empty, then the block pointed to by the free block pointer is allocated. The second hash function would ensure that the blocks would be sufficiently spilt. However, there may be a case when all the triples in the existing block would get mapped onto the same child block which is either the left block or the right block. The triple would not be inserted and the message *record_triple()*

would return to the calling message *add_triple()*. It would return a attribute *inserted* to the calling program with the value of 0 indicating that the triple has not been inserted and the address of the newly allocated block in attribute *newblk_address*. The tree structure is updated in the message *add_triple()* by storing the addresses of the left and right blocks at the left child location and right child location respectively of the existing node. The message *add_triple()* is executed again with the new tree structure which would again call the message *record_triple()*. This process would go on till the triple is inserted into the database. The second hash function has been designed so that these cases occur to the minimum [18].

The second hash function is applied to the key of the triple to be inserted and is inserted either into the left or the right block mapped block. The block header is updated accordingly and the blocks are stored at their corresponding addresses on secondary storage. The tree structure is updated as described above when the function returns back to the calling program leading to successful addition of the triple into the database.

In any case, the number of free blocks used is sent back and it is retrieved and stored in attribute *free_blk_used* in message *add_triple()*. This information is used to update the free-block list stored in object *free_blk_list* which is needed for further processing.

del_triple () is a C module that performs the deletion of the triple in the disk block. The parameters passed to *record_triple()* are as follows:

- *tbl_no* is as defined earlier.
- *text1*, *text2*, and *text3* is as defined earlier.
- *tbl_name* is as defined earlier.
- *blk_address* is as defined earlier.

For every triple to be inserted, the message *del_triple()* first reads in the main memory, the block pointed to by the *blk_address* parameter. It then checks if a triple with the exact match of its key exists in this block.

If true, it then checks if all the three fields of the triple to be deleted have the same match as the triple in the block. There may be either a single triple with the same key or a group of triples with lie in a chain of blocks. If all the fields of the triples do not match then the *next* field of the triple is checked to see if there exists any more triples with the same key. If there are no more triples, the function terminates with a message indicating that the triple does not exist. However, if a block address is stored at the *next* field of the triple, then the block is read into memory. Each and every triple of this block is then checked to find an exact match of the triple to be deleted. If the triple does not lie in this block then the next block on the chain, if any, is obtained by reading the *next* field of the block. This would go on until either the triple is found or end of the chain is reached. If the triple is found on the chain, the block header of the block which holds it is decremented by 1 and stored at its address with the changes.

There may be a case wherein the triple is the only one that exists in the block. In this case, the block would not hold any more triples on deletion of it's only triple. This block address of this free block is returned by the message *del_triple()*. This is retrieved in the calling message *delete_triple()* and the free block list is updated.

ret_triple() is a C module that performs the retrieval of triples from the disk block. It retrieves either a single triple or a group of triples. The tables from which the triples are to be retrieved depends on the key or the pattern of retrieval. The parameters passed to *ret_triple()* are as follows:

- *tbl_no* is as defined earlier.

- *text1*, *text2*, and *text3* is as defined earlier.
- *tbl_name* is as defined earlier.
- *blk_address* is as defined earlier.

The message *ret_triple()* first reads in the main memory, the block pointed to by the *blk_address* parameter. It then checks if a triple with a similar key exists in this block.

If true, it prints out the triple. There may either be a single triple or a group of triples having the same key. These triples may be held in blocks which are chained to each other. The *next* field in the triple gives the address of the next block on the chain. All the triples in this block are outputted. If there are any more blocks on this chain, then they are obtained by checking the *next* field of the block. This is illustrated in the block structure as shown in figure 5. This would go on till the end of the chain is reached.

ascii_eql() calculates the ascii equivalent of the key of the triple. It is a part of the first hash function that is applied to the triple before inserting, retrieving, or deleting it from the table. The hash function uses the modulo (mod) operator. The ascii equivalent of the key of the triple is divided by the size of the hash table and the remainder is used as a hash address. The hash address is the position of a cell on the hash table.

get_bit() is the second hash function that is applied to the triple before inserting, deleting, or retrieving it from the table. There are several methods for resolving collisions which occur when two or more triples hash to the same location. In this method, the key of the triple is converted to a stream of binary digits. This method is proposed by Frost[18] and consists of two stages:

1. The key is converted to an intermediate sequence of binary digits by the direct replacement of characters by the corresponding binary codes in the programmer defined table [Table 7]. This table is designed such that:

* Most frequently used characters are given binary codes with the most even distribution of 0's and 1's.
* Characters comprising of frequently used subsets, such as decimal digits and alphabetic characters , are given binary codes such that each subset as a whole has a reasonable even distribution of 0's and 1's.

2. The final binary stream is generated by picking bits off the intermediate stream as follows: the first bit of the stream corresponding to the first character is picked off, and then the first bit of the character corresponding to the second character, and so on. This process is then repeated for the second bit and so on. This process reduces the effect when two or more keys start with the same character.

These digits are then considered individually, and are used to determine paths in some binary tree which is constructed to hold the colliding records. *get_bit()* has the following parameters:

- *Key* is the key of the triple to be inserted, deleted, or retrieved from the database. It is a string of characters.
- *location* is the position of the character in the bit stream generated by the key of the triple. This value also corresponds to the current level of the tree at which the triple is to be inserted, retrieved, or deleted from the table.

The message *get_bit()* accepts the above parameters and returns the character at the position specified in attribute *location* from the bit stream generated by string specified in attribute *Key*.

Class HASH_TABLE1 is a specialized class known as the subclass of *class* BASIC_TABLE. This declares the methods for the first table out of the seven tables that make up the binary relational storage structure. This table uses the subject field of the

triple as the key for inserting, deleting, and retrieving a triple. It inherits all the attributes and methods of *class* BASIC_TABLE. It can also define additional attributes and methods thus leaving scope for specialized processing of the table defined in *class* BASIC_TABLE. It defines four methods which in turn use the methods inherited from its superclass BASIC_TABLE. These methods are *startup()*, *do_adding()*, *do_deleting()*, and *do_retrieving()*.

startup() performs operations to initialize the table. It invokes the method *initialize()*, inherited from *class* BASIC_TABLE, to perform the necessary operations.

do_adding() performs some preprocessing before adding a triple into the first table. It has the following parameters:

- *text1*, *text2*, and *text3* is as defined earlier.

It invokes the method *add_triple()*, inherited from *class* BASIC_TABLE, to perform the necessary operations.

do_deleting() performs some preprocessing before deleting a triple from the first table. It has the following parameters:

- *text1*, *text2*, and *text3* is as defined earlier.

It invokes the method *delete_triple()*, inherited from *class* BASIC_TABLE, to perform the necessary operations.

do_retrieving() performs some preprocessing before retrieving a single triple or a set of triples from the first table. It has the following parameters:

- *text1*, *text2*, and *text3* is as defined earlier.

It invokes the method *retrieve_triple()*, inherited from *class* BASIC_TABLE, to perform the necessary operations.

Class HASH_TABLE2 is identical to class HASH_TABLE1 and uses the relationship field of the triple as the key for inserting, deleting, and retrieving a triple.

Class HASH_TABLE3 is identical to class HASH_TABLE1 and uses the object field of the triple as the key.

Class HASH_TABLE4 is identical to class HASH_TABLE1 and uses a combination of the subject field and the relationship field of the triple as the key.

Class HASH_TABLE5 is identical to class HASH_TABLE1 and uses a combination of the relationship field and the object field of the triple as the key.

Class HASH_TABLE6 is identical to class HASH_TABLE1. It declares the methods for the sixth table which uses a combination of the subject field and the object field of the triple as the key.

Class HASH_TABLE7 is declares the methods for the seventh table which uses a combination of all the three fields of the triple as the key for inserting, deleting, and retrieving.

§ 5.2 Examples of code

The object-oriented paradigm has a lot of features which have enhanced the construction of the triple store. These features are as follows:

5.2.1 Dumping and retrieving the tree structure from the mainstore

Consider the following piece of code:

```
initialize (flname,tbl_name,env_key,free_key: STRING) is

  local
    i: INTEGER;

  do
```

```
camera.Create;
tablename.Create(flname);
camera.set_file(tablename);
camera.retrieve;
if camera.ok then
  tabl?=camera.item(env_key);
  from
    i:=1;
  until
    i>10
  loop
     --Create and assign the basic blocks to each index in the table
    aTree.Create(assign_initial_block(tbl_name.to_c));
    tabl.put(aTree,i);
    i:=i+1;
  end;
  camera.force(tabl,env_key);
  if not camera.ok then
    io.putstring("NOT RECORDED");
    io.new_line
  end;
  camera.close;
  camera.store;
  tablename.add_permission ("ugo","w");
 end;
end;
```

Here, *camera* is an environment variable. An environment is set of objects. Individual objects may be identified by a key with respect to an environment and these are the persistent objects of the environment. The complete object structures are stored in binary format into a file and can be retrieved very easily for later use. Any object referred directly or indirectly will also be copied.

Every table in this data structure has its own set of binary trees in the mainstore. These are dumped on the secondary storage using the Eiffel features *force* and *store* of class ENVIRONMENT. They can be retrieved by using the Eiffel feature *retrieve*. This simplifies the task of dumping and retrieving the tree structure owing to the fact that there is no need to write special purpose scanning programs such as those described in [18].

5.2.2 Use of class

A class is basically an implementation of an abstract data type. It describes a set of run-time objects, characterized by the features applicable to them, and by the formal properties of these features. The classes implemented in this system are as follows:

- **Class BASIC_TABLE** creates a table with the three basic operations to be performed on the triple. These operations are *initialize()*, *add_triple()*, *delete_triple()*, and *retrieve_triple()*.
- **Class INTERACTION** interacts with the user. It accepts the user input which may be either to add a triple, delete a triple, or retrieve a set of triples.

Besides this, there are seven other classes, one for each hash table. These classes interact with the other classes through the interface. Everything in the object-oriented paradigm revolves around this user-defined data type. Instead of being collections of independent functions connected by often complex sequences of program statements, programs tend to be clusters of classes. Each class, in turn, handles the complexity for its part of the program. As a result, the statements that are not part of any class are minimized and the connecting code often seems simple in comparison to the complicated system that is implemented.

C provides basic mechanisms, such as header files and separately compiled functions, to help programmers divide a program into modules that are functionally related, yet

separately maintainable. In a truly modular program design, the programmer needs to control how the modules are accessed and to determine which parts of each module are private and which are public. Through the mechanism of the class, the object-oriented paradigm provides a new degree of control for specifying the relationship among various objects in a program, as well as a way to specify how each class of objects is used.

5.2.3 Use of encapsulation within a class

One of the key characteristics of the object-oriented programming paradigm is the capability of encapsulating data and methods within a single entity. All the classes that are defined in this system have two regions. The public part gives access to the class; it allows other parts of the program a controlled entrance to the class and the class values. The private part of the class is unique to the class and is inaccessible except to members. The public part of the class, which usually consists of function members, represents a highly controlled access. Consider the class BASIC_TABLE.

```
class BASIC_TABLE export

    initialize,add_triple,delete_triple,retrieve_triple
```

The export clause defines the interface to the class. It specifies a set of features which may be available to all classes. The internal details of the class can change, and as long as the interface remains the same, these changes have no effect on the rest of the program. This is very useful for program maintenance and debugging. A program becomes a collection of autonomous entities. It becomes much easier to make changes or isolate obscure bugs.

5.2.4 Use of inheritance

One of the key points of object-oriented programming is that it allows for the creation

of hierarchies. A class object is used as the basis for defining other classes; these classes in turn can be used as the basis for another level of objects. Inherited code also enhances efficiency. It is no longer needed to create repetitive code. It is also no longer needed to create a series of independent and redundant data structures and functions to handle every possible variation of data organization used in a program. For example, the system defines a class BASIC_TABLE which defines the basic operations to be performed on the table. Seven specialized classes, one for each hash table, inherit the basic information from the class BASIC_TABLE.

The derived classes save the programmer a great deal of effort because blocks of redundant information don't need to be maintained. Parts that are unlikely to change in the foreseeable future are grouped together in the base class which can be derived for later use. This facilitates future modification. Using traditional methods, each time a field or memory is repeated, the memory allocation grows, and access time correspondingly slows. In current system, a lot of memory is saved by creating only one template of the table. The memory allocated for the hash table and the free block list need not be repeated. So, it is not necessary to create a complex set of conditionals to link one set of values to another. Inheritance provides an additional method of structuring the program leading to a more modular code.

5.2.5 Use of built-in data structures

Eiffel has a very comprehensive data structure library. It supports a number of fundamental data structures and algorithms like lists, queues, trees, stacks etc. This saves a lot of programming time and also prevents the code from being messy. It also improves the readability of the code.

§ 5.3 Estimates of performance

The performance estimates of the binary-relational storage structure can be summarized as follows :

- A single triple requires only one access to the backing store. To retrieve say N triples with equal key, it would require (N/B) +1 access, where B is the no of triples that are held in a block. Triples with the same key are chained together in blocks on the secondary storage. This is a very important feature of the data structure. The actual access time will depend upon the disk technology used.
- The system has been built using the Eiffel programming language. Eiffel is a strongly typed language using static typing based on dynamic binding. Dynamic binding results in costly run-time searches. This is because a requested routine may not be defined in the class of the target object but inherited from a possibly remote ancestor. Static typing will enable a good implementation to find the appropriate routine in constant-bounded time.

Chapter 6 CRITICAL ANALYSIS OF THE USE OF THE OBJECT-ORIENTED PARADIGM IN THE CONSTRUCTION OF A TRIPLE STORE BASED ON DYNAMIC HASHING

Almost always, new software expands on previous developments; the best way to create it would seem to be by imitation, refinement, and combination. But most design methods generally ignore this aspect of system development. This is the most important concern in an object-oriented approach. The major features of using the object-oriented paradigm discussed below which summarize material presented in [26]:

§ 6.1 Object-oriented code structure facilitates construction, debugging and maintenance

Object-oriented programming is a structuring technique. In object-oriented programming, objects are the principle elements of construction. However, simply understanding what an object is or using objects in a program does not mean that the object-oriented paradigm is being used. It depends on the way in which the objects are put together.

In procedural programming, it is possible to build a program out of procedural elements and still violate the principles of sound construction. Simply knowing what a procedure is or how to build one does not guarantee a robust, bug-free program that is easy to use, easy to read, and easy to modify. The same is true for an object-oriented approach. However, object-oriented principles such as polymorphism, inheritance, and encapsulation enhance the design of the system.

Consider a procedural-program structure that advocates hiding information as much as possible to keep the components of a program modular. Local data is hidden inside of procedures, and information that needs to be shared by procedures is passed in the form of procedural arguments. Sometimes, however, the number of procedures that need to

share information is too large, or the data structures themselves are too large to be passed between procedures. In such cases, global data is used. The problem with global data is that it can be easily altered by any procedure within the program. This makes it easier for bugs to creep in, and it makes it difficult to track down the bugs because the scope of the statements that could possibly cause an observed problem is now the entire program. In an object-oriented environment, objects are the primary structural units. The object module is divided into a public interface and a private part. Only procedures within the object are allowed to access the private data elements. This ensures that the data is not accidentally altered by procedures outside the structural boundary. If erroneous data is found when debugging the structure, it can be immediately traced to one of the procedures that have access to the data.

Object-oriented programming provides a way to structure and manage complex relationships among a large number of system components. Forcing objects to communicate through a narrow public interface makes it easier to isolate bugs and determine which elements are responsible for the bugs that do occur.

The triple store has seven versions of a table which are slightly different from each other. The common features are the initialization of the table, addition of a triple into the database, deletion of a triple from the database, and retrieval of a set of triples from the database. These are performed by methods *initialize()*, *add_triple()*, *delete_triple()*, and *retrieve_triple()* respectively. These stable features of the table are captured in a class known as the base class. In this case, the class BASIC_TABLE contains the common and the stable features of the all the tables. The other tables inherit these features from the class BASIC_TABLE in order to do their own specialized processing. Consider the following piece of code from the method *add_triple()* of class BASIC_TABLE:

```
camera.Create;

tablename.Create(flname);
camera.set_file(tablename);
camera.retrieve;
if camera.ok then
  from
    camera.open;
    inserted:=0;
    level:=0;
    pos:=ascii_eql(tpl_key) mod 10;
    if pos=0 then
      pos:=10;
    end;
    tabl?=camera.item(env_key);
    free_blk_list?=camera.item(free_key);
    check_string.Create(15);
  until
    inserted=1
  loop
    from
      aTree:=tabl.item(pos);
    until
      aTree.item /= -1
    loop
      level:=level+1;
      check_bit:=get_bit(tpl_key,level);
      if check_bit='0' then
        aTree:=aTree.left_child
      end;
      if check_bit='1' then
        aTree:=aTree.right_child
      end;
    end;
```

```
blk_address:=aTree.item;

free_count:=free_blk_list.count;
```

The above code represents the tree scanning procedure before insertion of a triple into the database. It scans the tree from the root node until it reaches the leaf node which contains the address of the block where the triple is to be inserted. This procedure is a useful tree-scanning mechanism which would be common for all the tables in this kind of structure. Owing to the fact that the base class contains features which are less prone to change, and also that the other classes can inherit from the base class, makes the overall program design highly structured.

Inheritance is one of the key perspective that helps in the structuring of code.

§ 6.2 Inhertitance facilitates reuse and maintainence of code

Classes can be treated as both modules and types. Inheritance brings light to both these viewpoints which can be considered as follows:

2.1 The module perspective

Inheritance is an important reusability technique from the module viewpoint. A module is a set of services offered to the outside world. Without inheritance every new module must itself define all the services it offers. The implementations of these services may rely on the services provided by the other modules. But there is no way to define a new module as simply adding new services to previously defined modules.

Inheritance provides this possibility. If B "inherits" from A, then all the features of A are automatically available in B, without any need to further define them. B is free to add any new features for its own specific purposes. An extra degree of flexibility is provided by "redefinition", which allows B to take its pick from the implementations offered by A: some may be kept as they are, whereas others may be overridden by more

appropriate ones local ones. This style of software development is quite different from the traditional approaches. Instead of trying to build from scratch, the idea is to build on previous accomplishments and extend their results. For example, in the current system, class BASIC_TABLE defines four basic operations. They are *initialize()*, *add_triple()*, *delete_triple()*, and *retrieve_triple()* defined as follows:

```
initialize (flname,tbl_name,env_key,free_key: STRING) is
local
  i: INTEGER;
do
  camera.Create;
  tablename.Create(flname);
  camera.set_file(tablename);
  camera.retrieve;
  if camera.ok then
    tabl?=camera.item(env_key);
    from
      i:=1;
    until
      i>10
    loop
       --Create and assign the basic blocks to each index in the table
      aTree.Create(assign_initial_block(tbl_name.to_c));
      tabl.put(aTree,i);
      i:=i+1;
    end;
    camera.force(tabl,env_key);
    if not camera.ok then
      io.putstring("NOT RECORDED");
      io.new_line
    end;
    camera.close;
    camera.store;
```

```
    tablename.add_permission ("ugo","w");
  end;
end;


add_triple(tbl_no: INTEGER; flname,env_key,free_key,
           tbl_name,tpl_key,text1,text2,text3: STRING) is
  local
    i,j,pos,blk_address,inserted,free_blk_used,free_count,
    level,newblk_address: INTEGER;
    check_string,check_substr: STRING;
    check_bit: CHARACTER;
    temp_list: ARRAY[INTEGER];
  do
    camera.Create;
    tablename.Create(flname);
    camera.set_file(tablename);
    camera.retrieve;
    if camera.ok then
      from
        camera.open;
        inserted:=0;
        level:=0;
        pos:=ascii_eql(tpl_key) mod 10;
        if pos=0 then
          pos:=10;
        end;
        tabl?=camera.item(env_key);
        free_blk_list?=camera.item(free_key);
        check_string.Create(15);
      until
        inserted=1
      loop
        from
```

```
  aTree:=tabl.item(pos);
until
  aTree.item /= -1
loop
  level:=level+1;
  check_bit:=get_bit(tpl_key,level);
  if check_bit='0' then
    aTree:=aTree.left_child
  end;
  if check_bit='1' then
    aTree:=aTree.right_child
  end;
end;
blk_address:=aTree.item;
free_count:=free_blk_list.count;
check_string:=record_triple(free_blk_list.to_c,free_count,text1.to_c,
                            text2.to_c,text3.to_c,tbl_name.to_c,
                            blk_address,level,tbl_no);
check_substr:=check_string.substring(1,1);
inserted:=check_substr.to_integer;
check_substr:=check_string.substring(2,2);
free_blk_used:=check_substr.to_integer;
check_substr:=check_string.substring(3,check_string.count);
newblk_address:=check_substr.to_integer;
temp_list.Create(1,(free_count-free_blk_used));
from
  i:=1;
  j:=free_blk_used+1;
until
  i>(free_count-free_blk_used)
loop
  temp_list.put(free_blk_list.item(j),i);
  i:=i+1;
  j:=j+1;
```

```
        end;
        free_blk_list.wipe_out;
        free_blk_list:=temp_list;
        if newblk_address /= -1 then
          aTree.put(-1);
          aTree.child_put_left(blk_address);
          aTree.child_put_right(newblk_address);
        end;
        aTree.child_start;
        tabl.force(aTree,pos);
      end;
      camera.force(tabl,env_key);
      camera.force(free_blk_list,free_key);
      if not camera.ok then
        io.putstring("NOT RECORDED");
        io.new_line;
      end;
      camera.close;
      camera.store;
      tablename.add_permission("ugo","w");
    end
  end;

delete_triple(tbl_no: INTEGER; flname,env_key,free_key,
              tbl_name,tpl_key,text1,text2,text3: STRING) is
  local
    pos,level,blk_address,check_address,side,free_blk_ctr,value: INTEGER;
    check_bit: CHARACTER;
  do
    camera.Create;
    level:=0;
    tablename.Create(flname);
    camera.set_file(tablename);
    camera.retrieve;
```

```
if camera.ok then
  pos:=ascii_eql(tpl_key) mod 10;
  if pos=0 then
    pos:=10;
  end;
  tabl?=camera.item(env_key);
  free_blk_list?=camera.item(free_key);
  camera.open;
  side:=-1;
  from
    aTree:=tabl.item(pos)
  until
    aTree.item /= -1
  loop
    level:=level+1;
    check_bit:=get_bit(tpl_key,level);
    if check_bit='0' then
      aTree:=aTree.left_child;
      side:=0
    end;
    if check_bit='1' then
       aTree:=aTree.right_child;
       side:=1
    end;
  end;
  blk_address:=aTree.item;
  check_address:=del_triple(tbl_no,text1.to_c,text2.to_c,text3.to_c,
                            tbl_name.to_c,blk_address);
  if check_address /= -1 then
    if level /= 0 then
      free_blk_ctr:=free_blk_list.count;
      free_blk_ctr:=free_blk_ctr+1;
      free_blk_list.force(check_address,free_blk_ctr);
      aTree:=aTree.parent;
```

```
          if side=0 then
            aTree:=aTree.right_child;
            value:=aTree.item;
            aTree:=aTree.parent;
            aTree.remove_left_child;
            aTree.remove_right_child;
            aTree.put(value);
          end;
          if side=1 then
            aTree:=aTree.left_child;
            value:=aTree.item;
            aTree:=aTree.parent;
            aTree.remove_left_child;
            aTree.remove_right_child;
            aTree.put(value);
          end;
        end;
      end;
      tabl.put(aTree,pos);
      camera.force(tabl,env_key);
      camera.force(free_blk_list,free_key);
      if not camera.ok then
        io.putstring("NOT RECORDED");
        io.new_line;
      end;
      camera.close;
      camera.store;
      tablename.add_permission("ugo","w");
    end;
  end;
retrieve_triple(tbl_no: INTEGER; flname,env_key,tbl_name,
                tpl_key,text1,text2,text3: STRING) is
  local
    pos,blk_address,inserted,level,newblk_address: INTEGER;
```

```
  check_bit: CHARACTER;
do
  camera.Create;
  level:=0;
  tablename.Create(flname);
  camera.set_file(tablename);
  camera.retrieve;
  if camera.ok then
    pos:=ascii_eql(tpl_key) mod 10;
    if pos=0 then
      pos:=10;
    end;
    tabl?=camera.item(env_key);
    from
      aTree:=tabl.item(pos);
    until
      aTree.is_root=true
    loop
      aTree:=aTree.parent;
    end;
    from
    until
      aTree.item /= -1
    loop
      level:=level+1;
      check_bit:=get_bit(tpl_key,level);
      if check_bit='0' then
        aTree:=aTree.left_child;
      end;
      if check_bit='1' then
         aTree:=aTree.right_child;
      end;
    end;
    blk_address:=aTree.item;
```

```
    ret_triple(tbl_no,text1.to_c,text2.to_c,text3.to_c,
               tbl_name.to_c,blk_address);
  end;
end;
```

These functions are inherited by the subclasses or derived classes of class BASIC_TABLE. So they need not be redefined in every class. As stated earlier, the derived classes save the programmer a great deal of effort because blocks of redundant code don't need to be maintained. Using traditional methods, each time a field or memory is repeated, the memory allocation grows, and access time slows down correspondingly. In the current system, memory is saved by creating only one template of the table. The memory allocated for the hash table and the free-block list need not be repeated. So, it is not necessary to create a complex set of conditionals to link one set of values to another.

2.2 The type perspective

From the type perspective, inheritance addresses both reusability and flexibility. It does this with dynamic binding. A type is basically a set of values characterized by certain operations. Consider the class BASIC_TABLE. It can be considered as a type of object that initializes a table and performs operations such as adding a triple into a database, deleting a triple from the database, and retrieving a set of triples from the database. This class can be inherited by other classes with some more operations added to the existing ones. Thus the concept of reusability defines a new type which is a subclass of the existing class. Now consider an assignment statement of the form `a:=b`. Suppose *a* is an instance of class A and *b* is an instance of class B. Now, the instance *b* can either be of the same type as that of class A or it can be a subclass of class A. The type of the objects can be determined dynamically at runtime. This gives flexibility to the systems. The following example illustrates this point:

Consider a class POLYGON which defines a feature called *perimeter* which computes its perimeter. Class RECTANGLE, an heir of class POLYGON, inherits this feature and redefines it. Other descendants of class polygon may also have their own redefinitions of perimeter. Now, consider the following piece of code: `p: POLYGON; r: RECTANGLE`

```
.....
/* Initialize class POLYGON and class RECTANGLE
p.Create; r.Create;
...
if c then p:=r end;
print (p.perimeter)
```

As seen above, the computation of *p.perimeter* would depend upon the value of *c*. If *c* is false *p* will be attached to an object of type POLYGON else it will be attached to an object of type RECTANGLE and will execute the redefined version of *perimeter*.

§ 6.3 Specialization helps in structuring

Inheritance can be viewed as a specialization when a class is viewed as a type. For example, class HASH_TABLE1 is a more specialized version of class BASIC_TABLE. If B is an heir to A, the objects that may be associated at run-time with an entity of type B form a subset of the set of objects that may be associated with an entity of type A. This helps in maintenance and also makes the code structured.

§ 6.4 Objects facilitate program modification and extension

From the module perspective, a class is viewed as a provider of services, B implements the services of A plus its own. So in our system, class BASIC_TABLE provides services to the seven subclasses which correspond to each hash table. In addition to this, each and every table not only inherits the features of class BASIC_TABLE but also adds some features. So this structure is easily extensible and facilitates modification. The

programmer can conceive a new table based the existing classes and keep the structure modular and maintainable.

Consider a few more examples to illustrate this point. Every triple before being inserted, deleted, or retrieved from the database has to go through two hash functions. The first hash function maps it to a cell on the hash table and the second hash function performs the scanning of the binary tree to reach a leaf node that contains the block address. Now the second hash function, known as *get_bit()*, is defined in class BASIC_TABLE. This function is invoked with the following message structure:

```
<object_name>.get_bit()
```

To the other members, the implementation details of the second hash function are transparent. So the internal details of the function can easily be modified without causing any structural changes in the system. So as stated earlier, the object module is divided into a public interface and a private part. Only procedures within the object are allowed to access the private data elements. This ensures that the data is not accidentally altered by procedures outside the structural boundary.

Another example, is the tree scanning process during insertion, deletion, or retrieval of triples. This procedure is defined within the methods *add_triple()*, *delete_triple()*, and *retrieve_triple()* of class BASIC_TABLE. If however, the tree scanning process were changed to incorporate some more features such as backtracking to reach to the leaf node, it would not cause any effect on the implementation of the modules.

Consider the method *delete_triple()*. In the current implementation of the system, only a single triple can be deleted from the database. If the system were extended to incorporate multiple deletions, it would not affect the overall structure of the system. The method *delete_triple()* would undergo a few changes. It would first check if it is a single deletion or a multiple deletion. If it is a single deletion then the normal deletion

process would be done which would delete the triple from all the seven tables. For multiple deletion, the triples would first be deleted from the table corresponding to the key. For example, consider the deletion of the following set of triples viz. `(ibm, ?, ? )`. This query specifies the deletion of all triples whose subject identifier field contains "ibm". To execute this query, the system would inspect the first hash table and pick out all triples whose subject identifier contains "ibm". These triples would be stored in a temporary storage and deleted from the first hash table. Based on their respective keys, the remaining hash tables would then be checked for each and every triple in the temporary structure and deleted from the database. Thus these triples would then be deleted from the remaining six tables. Finally, this temporary structure would also be disposed. All of these changes would affect the internal implementation of the method and would cause no changes to the interface, thus isolating the implementation details of the method from the rest of the system. These examples illustrate how the object-oriented paradigm facilitates maintenance.

It is also very easy to incorporate changes to view the internal details of the tree structure such as the depth of a tree or number or nodes. Eiffel supports a comprehensive library of fundamental data structures such as trees, lists, stacks etc., covering many of the needs of common programming applications. This system uses the built-in binary tree structure. This structure has a number of features some of which are highlighted below:

- ***arity*** : This checks the number of nodes in the tree.
- ***is_leaf*** : This checks whether the existing node is a leaf node.
- ***is_root*** : This checks whether the existing node is a root node.
- ***height*** : This feature gives the height of the tree.
- **parent** : This gives the parent of the current node.
- ***has_left*** : This checks if the left child exists or not.

- ***has_right*** : This checks if the right child exists or not.
- ***has_none*** : This checks whether the current node is childless.

These features make it flexible for the programmer to view the internal details of the file.

§ 6.5 The object-oriented paradigm facilitates the maintenance of database consistency

The object-oriented paradigm provides a convenient way of maintaining the consistency during multiple updates. The dynamic-hashing triple store data structure is highly prone to inconsistency during insertion, deletion, and retrieval of triples. For example, the triple has to be inserted in all the seven tables. During such insertion, if there is a power failure or a disk crash, the system must be able to perform an appropriate action or it would lead to an inconsistency in the database. The object-oriented paradigm helps to solve this problem. The following is one 'object-oriented' solution to the problem that would not cause any structural changes to the system and would still keep the code maintainable:

Each and every triple in the database can be treated as an object. These triples are stored in a block which would also be treated as an object. Insertion of a triple would involve updating seven different tables. To incorporate these changes, the method *add_triple()* would initially scan the binary tree and locate the appropriate block address on the secondary storage. A copy of the new updated block would be stored in a temporary structure but these changes would not be incorporated in the actual table. If the block is to be spilt into two then the new blocks would be stored in the temporary structure without changing the original tree structure. This process would be executed for each and every table. Finally, when the database commits to update, these objects

stored in the temporary structure, would then be attached at the appropriate positions for every table. The temporary structure would then be disposed to free the memory that it occupies. This would be more stable than before, as the database would rollback to its original state if an error occurs. Also update time would be minimized as the time taken to scan the tree and locate the appropriate block would be eliminated for every table. It would just be a matter of updating the links in every table that are stored in the temporary structure.

However, the current implementation of Eiffel, on which the system has been developed does not have a very good interface to the Unix file system. This has been improved in the latest version which is already in use in the industry. The current implementation uses the C language to perform the low-level disk operations. So updating the blocks would have additional overhead such as maintaining an additional array which would hold the new information then updating it in the original table on receiving a commit message. In this case all the processing would be done centrally in the method i.e. *add_triple()*. It would not break up into further objects and communicate using the message passing mechanism. This would have an additional effect on the modularity of the system. However, the fact that these implementation details are encapsulated within the module and are hidden from others, the extension would not affect the overall modularity of the system.

§ 6.6 The object-oriented paradigm facilitates storage and re-loading of dynamically-created mainstore data structures

The object-oriented paradigm allows the storage of complete object structures in a file in binary format. The stored object structures can be retrieved with ease at the beginning of a new session. The objects are stored in an environment. An environment is a set of objects. Individual objects may be identified by a key with respect to an environment.

Such objects, and all their direct and indirect dependants are the persistent objects of the environment. In our system, an environment object called *camera* is defined in the class BASIC_TABLE.

When an environment is opened all objects created thereafter belong to the environment until it is closed. Storing is achieved either explicitly (through the *store* procedure) or implicitly at session termination (if the system executed *store_on_end* or *store_on_failure*). The class BASIC_TABLE uses the *store* procedure. Both store and retrieve use the file associated with an environment by the *set_file* procedure. Eiffel provides these features through the class ENVIRONMENT. This enables a database which is created or modified during one session to be easily re-loaded for further use.

§ 6.7 The message passing mechanism facilitates modularity

Objects communicate to one another by sending messages. They send and receive information from one another by sending messages. Messages have a counterpart called methods. Methods are the procedures that are invoked when an object receives a message. For example in the current system, class INTERACTION defines an object for each hash table. This is shown below:

```
htabl1: HASH_TABLE1;

htabl2: HASH_TABLE2;
htabl3: HASH_TABLE3;
htabl4: HASH_TABLE4;
htabl5: HASH_TABLE5;
htabl6: HASH_TABLE6;

htabl7: HASH_TABLE7;
```

Each of these hash tables performs its operations through messages. For example to perform certain startup operations the following message is performed by each table:

```
htabl1.startup;

htabl2.startup;
htabl3.startup;
htabl4.startup;
htabl5.startup;
htabl6.startup;

htabl7.startup;
```

Similarly, each table performs the addition, deletion, and retrieval operations through messages *do_adding* , *do_deleting*, and *do_retrieving* respectively. In traditional programming terminology, a message is a call from one procedure to another. A method is the actual code or procedure definition that is invoked. The methods defined in each of the hash tables in the triple-store are *do_adding*, *do_deleting*, and *do_retrieving*. These methods in turn call *add_triple()*, *delete_triple()*, *retrieve_triple()* with their parameters. The methods *add_triple()*, *delete_triple()*, *retrieve_triple()*, that are used in the individual hash table, are inherited from the class BASIC_TABLE. At first, this terminology appears arbitrary. That is because traditional programming languages do not support polymorphism or message dispatching handled by objects themselves at run time.

"Methods and messages help enforce division of labor. A method is carried out by the receiving object in response to a message. Messages provide communication through narrow bandwidth interfaces to objects" [30]. Message passing reduces the number of connections among system components. Reducing the number of connections increases the modularity of system components. Messages carry information. They do so through arguments and return values, just as procedures do.

§ 6.8 Difficulties of using the object-oriented paradigm

The disadvantages of using the object-oriented paradigm are as follows:

- There are no formal semantics available for the object-oriented paradigm.
- We cannot prove anything with certainty because implementation of the programs could be anything.

Chapter 7 ADVANTAGES/DISADVANTAGES OF USING EIFFEL

§ 7.1 Specific advantages of Eiffel

The features of the Eiffel programming language can be summarized as follows:

- It has a library of built in data structures. The data structure library contains implementations of fundamental data structures and algorithms such as lists, trees, stacks, queues and many others. When the classes that belong to the system are compiled, the system contained in the current directory produces an interactive system that lets one build, explore and modify interactively instances of many of the fundamental data structures implemented by the library.
- It supports an exception handling mechanism. Exceptions or contract violations may arise from several causes. One is assertion violations if assertions are monitored. Another is the occurrence of a signal triggered by the hardware or operating system to indicate an abnormal condition such as arithmetic overflow or lack of memory to allocate a new object. Unless a routine has made specific provision to handle exceptions, it will fail if an exception arises during an execution. A routine handles an exception through a rescue clause.
- It supports the concept of feature redefinition. A class may redefine some or all of the features which it inherits from its parents. For an attribute or function, the redefinition may affect the type, replacing the original by a descendant; for a routine it may also affect the implementation, replacing the original's routine body by a new one.
- Eiffel is a strongly typed language. It has static typing, polymorphism with dynamic binding. The power of polymorphism and dynamic binding demand adequate controls. Because the language is typed, a compiler can check statically whether a feature application *a.f* is correct. In contrast, dynamically typed object-oriented

languages defer checks until run-time. If an object "sends a message" to another, one just expects that the corresponding class, or one of its ancestors will happen to include an appropriate routine; if not a run-time error will occur. Such errors may not happen during the execution of a type-checked Eiffel system. Dynamic binding guarantees that whenever more than one version of a routine is applicable the *right* version will be selected. Static typing means that the compiler makes sure there is *at least one* such version.

- It has a standardized set of carefully designed libraries. These are as follows:
  - The Kernel Library includes essential classes which are of use to many Eiffel applications, and to provide immediate extensions to the language. The classes described here are arrays, strings, file, and classes such as bool, char, int, float that serve as base classes for the basic types boolean, character, integer, and real.
  - The Support Library provides a number of classes that extend the facilities of the language and environment, enabling programmers to develop advanced uses of Eiffel. These include classes such as memory, environment, ascii etc. which handle persistence storage, memory management and the ASCII character set.
  - The Winpack library classes support the development of multi-windowing applications on character-oriented terminals.
  - The Data Structure library, as described above, has a library of built in data structures. The data structure library contains implementations of fundamental data structures and algorithms such as lists, trees, stacks, queues and many others.
  - The Iterator library is a set of iterator classes that can be used instead of control structures. These primitives apply to traversable data structures.
  - A Lexical library is a set of classes that make it possible to define a regular grammar and build a lexical analyzer for it. The Parsing library classes support

the object-oriented development of systems which must parse the structure of documents such as programs or specifications.

- It has an incremental garbage collector. The collector is organized as a coroutine and disturbs the main computing process as little as possible. The garbage collector uses an incremental algorithm which regularly collects unused space.
- It has a relationship to C and subsequent interface with the operating system for disk access.

The advantages of these particular features in the development of this system are as follows:

- The comprehensive library set of Eiffel makes it easy for the programmer. Class BASIC_TABLE uses a number of built-in features of the data structure library such as binary trees and arrays. These classes have features such as *arity*, *height* which makes it flexible for the programmer to view the internal details of the tree. This is a piece of code from the class BASIC_TABLE used to declare a hash table, free block list, and an environment variable.

```
tabl: ARRAY[BINARY_TREE[INTEGER]];
aTree: BINARY_TREE[INTEGER];
free_blk_list: ARRAY[INTEGER];
camera: ENVIRONMENT;
tablename: FILE;
```

- Eiffel and C routines can be combined in an application. The most common way of interfacing Eiffel and C is to allow an Eiffel application to call C functions declared as Eiffel external routines. As the current implementation of Eiffel does not have a very flexible interface to the Unix operating system calls, this interface is being used to serve this purpose. Class BASIC_TABLE defines four functions which call a C

function in turn. They are as follows:

- ***assign_initial_block()*** initializes the blocks on the secondary storage at the begining of a session.

```
assign_initial_block(t_name: INTEGER): INTEGER is
   external
      assign_initial_block(t_name: INTEGER): INTEGER
name "assign_initial_block" language "C"
   do
      Result:=assign_initial_block(t_name);
   end;
```

- ***record_triple()*** inserts the triple at a given block address in the secondary storage.

```
record_triple(free_blk_list,free_count,t1,t2,t3,
              t_name,blk_address,level,t_no: INTEGER): STRING is
   external
      record_triple(free_blk_list,free_count,t1,t2,t3,
                    t_name,blk_address,level,t_no: INTEGER): STRING
                    name "record_triple" language "C"
   do
     Result:=record_triple(free_blk_list,free_count,t1,t2,t3,
                           t_name,blk_address,level,t_no);
   end;
```

- ***delete_triple()*** deletes the triple from a given block address on secondary storage.

```
del_triple(tbl_no,t1,t2,t3,t_name,blk_address: INTEGER): INTEGER is
   external
      del_triple(tbl_no,t1,t2,t3,t_name,blk_address: INTEGER): INTEGER
                 name "del_triple" language "C"
   do
     Result:=del_triple(tbl_no,t1,t2,t3,t_name,blk_address);
   end;
```

- ret_triple() retrieves a single triple or a set of triples from a given block address

on secondary storage.

```
ret_triple(tbl_no,t1,t2,t3,t_name,blk_address: INTEGER) is
   external
      ret_triple(tbl_no,t1,t2,t3,t_name,blk_address: INTEGER)
                name "ret_triple" language "C"
   do
     ret_triple(tbl_no,t1,t2,t3,t_name,blk_address);
   end;
```

- Eiffel allows the storage of complete object structures in a file in binary format. The stored object structures can be retrieved with ease at the beginning of a new session. The objects are stored in an environment. An environment is a set of objects. Individual objects may be identified by a key with respect to an environment. Such objects, and all their direct and indirect dependants are the persistent objects of the environment. The following piece of code from class BASIC_TABLE demonstrates the initialization of a table on the mainstore. It then stores this table on secondary storage which acne be retrieved for later use.

```
initialize (flname,tbl_name,env_key,free_key: STRING) is

 local
   i: INTEGER;

 do
   camera.Create;
   tablename.Create(flname);
   camera.set_file(tablename);
   camera.retrieve;
   if camera.ok then
     tabl?=camera.item(env_key);
     from
```

```
        i:=1;
      until
        i>10
      loop
         --Create and assign the basic blocks to each index in the table
        aTree.Create(assign_initial_block(tbl_name.to_c));
        tabl.put(aTree,i);
        i:=i+1;
      end;
      camera.force(tabl,env_key);
      if not camera.ok then
        io.putstring("NOT RECORDED");
        io.new_line
      end;
      camera.close;
      camera.store;
      tablename.add_permission ("ugo","w");
    end;
  end;
```

§ 7.2 Specific difficulties of Eiffel

Although Eiffel has a lot of advantages, it has a lot of disadvantages which can be summarized as follows:

- It does not support extension polymorphism. A fixed signature is needed for every interface.
- Eiffel version 2.3 does not have a well-developed Unix library file for system calls.
- Eiffel has a 4–pass compiler. Hence it takes a long time to compile.
- An environment is a set of objects. Individual objects may be identified by a key with respect to an environment. Such objects, and all their direct and indirect dependents

are the persistent objects of the environment. An environment may be opened, stored, and retrieved. However, it is not possible to add into an existing environment file.

- Eiffel has an interface to C. However, the C interface is not very flexible. Structures in C cannot be passed as parameters.
- There are no proper debugging tools between C and Eiffel.

Chapter 8 CONCLUSIONS

§ 8.1 Analysis and results

The thesis to be defended in this report is :

> The object-oriented paradigm is well suited for the construction of a triple store based on dynamic hashing.

The work undertaken that relates to the thesis includes the following:

- A literature survey on *Object-Oriented Database Management Systems*. It was observed that most of the object-oriented database systems were built either on relational database systems or by using the feature of persistence in object-oriented programming languages.
- The author familiarized himself with the object-oriented paradigm. He developed skills in the object-oriented languages : C++, GNU Smalltalk, and Eiffel.
- The author built the triple-store system using the pure object-oriented language Eiffel and C.
- The use of the object-oriented paradigm in this application was analyzed.

The conclusions that can be drawn are:

- The analysis of the code for the triple-store supports the claim that the object-oriented paradigm facilitates software structuring for design and reuse.

  Although the code for the triple-store was not reused in another application some of the code was easily reused within the application. Because the triple store requires seven different versions of the data, the object-oriented paradigm is definitely well-suited to this aspect of the application.

- Although the triple-store code was not extended in this investigation, an analysis of what would be required if the code were to be modified to ensure minimal commit window (and therefore improve consistency) was carried out. This analysis shows how the object-oriented paradigm would facilitate such a modification of the code.

- The object-oriented code structure facilitates construction, debugging and maintenance owing to the fact that object-oriented programming provides a way to structure and manage complex relationships among a large number of system components. Forcing objects to communicate through a narrow public interface makes it easier to isolate bugs and determine which elements are responsible for the bugs that do occur.

- The object-oriented paradigm facilitates the maintenance of database consistency by providing an 'object-oriented' solution to the problem of minimizing the update commit window. The solution does not cause any structural changes to the system and does not affect the maintainability of the code.

- The message-passing mechanism facilitates modularity as message passing reduces the number of connections among system components and reduces the number of connections, which increases the modularity of system components.

- Eiffel has built-in functions which facilitate the storage of complete mainstore object structures in a file in binary format which could be retrieved with ease. This simplifies the task of dumping and retrieving the tree structure owing to the fact that there is no need to write special-purpose scanning programs. Eiffel's built-in functions are consistent with the object-oriented paradigm.

- The current implementation of Eiffel did not have a very flexible support for manipulating disk blocks on secondary storage. However, this can be improved with the new release of Eiffel.

- The interface with C gave a procedural look to the system, thereby allowing the programmer to slightly drift from the object-oriented principles.
- Analysis of the code reveals that an increased use of the interface to C could increase the complexity of the system, thereby raising concern about the object-oriented nature of the application.
- Correctness of the implementation cannot be proved or determined with certainty as the object-oriented paradigm lacks formal semantics.

§ 8.2 Future Work

During the past several years, the object-oriented approach to programming and designing complex software systems has received tremendous attention in the programming, knowledge representation, and database disciplines. The object-oriented concepts may be the key to building a high-performance programming systems in the foreseeable future. The object-oriented paradigm is beginning to be used in areas such as the following :

- ***Design databases :*** Engineering-design databases are useful in computer-aided design/manufacturing systems. In such systems, complex objects can be recursively partitioned into smaller objects. Furthermore, an object can have different representations at different levels of abstraction. Moreover, a record of the object's evolution should be maintained and this is done through object versions. This is a suitable application of the object-oriented paradigm as traditional database systems do not support the notions of complex objects.
- ***Multimedia software :*** In any multimedia system, data includes not only text and numbers but also images, graphics and digital audio and video. Such multimedia data are long and unstructured and are typically stored as sequence of bytes of variable lengths. Applications require access to this data on the basis of the structure of a

graphical item or by following a logical link. Conventional query languages would not be able to handle this.

- ***Artificial Intelligence or Expert systems :*** Artificial intelligence and expert systems [23] represent information as facts and rules that can be viewed as a knowledge base. In an AI application, knowledge representation requires data structures with rich semantics. Furthermore, operations in a knowledge base are complex. When a rule is added, the system must check for contradiction and redundancy. Such operations cannot be represented by relational operations and the complexity of checking increases rapidly as the size of the knowledge base grows.

It will be important to analyze the real value of the object-oriented paradigm in such applications through investigations such as the one described in this thesis. Such future work will help to separate the reality from the myths surrounding the object-oriented paradigm.

Appendix : A Program listing (s)

§ A.1 Class INTERACTION

```
-- Accept input from the users

class INTERACTION export


feature

  text1,text2,text3: STRING;
  htabl1: HASH_TABLE1;
  htabl2: HASH_TABLE2;
  htabl3: HASH_TABLE3;
  htabl4: HASH_TABLE4;
  htabl5: HASH_TABLE5;
  htabl6: HASH_TABLE6;
  htabl7: HASH_TABLE7;
  quit,flag,finish: BOOLEAN;
  test_file: FILE;

Create is

  do
    test_file.Create("Tree.env");
    if test_file.exists then
      flag:=true
    end;
    create_tables;
    if flag=false then
      io.new_line;
```

```
      io.putstring("***** INITIALIZING *****");
      io.new_line;
      do_initialization;
    end;

    get_choice;

  end;

get_choice is
  --Prompt the user to add, delete, or check a triple

  local
    correct:BOOLEAN;
    answer:CHARACTER;

  do
    from
      finish:=false
    until
      finish=true
    loop
      from
        correct:=false
      until
        correct
      loop
        io.putstring("Enter your choice (one character)");
        io.putstring("  (A)dd, (D)elete, (R)etrieve, (Q)uit : ");
        io.input.readchar;
        answer:=io.input.lastchar;
        io.input.next_line;
        correct:=true;
```

```
        inspect
          answer
        when 'a', 'A' then
          do_add
        when 'd', 'D' then
          do_delete
        when 'r', 'R' then
          do_retrieve
        when 'q', 'Q' then
          finish:=true
        else
          io.new_line;
          io.putstring("Incorrect code. Please enter again.");
          io.new_line;
          correct:=false
        end;
      end;
    end;
  end;

do_add is
        --Accept a triple from the user and add it to the database
  do
    from
      quit:=false;
    until
      quit=true
    loop
      io.putstring("\nEnter the first part of the triple : ");
      io.readline;
      text1 := io.laststring.duplicate;
      io.putstring("\nEnter the second part of the triple : ");
      io.readline;
      text2 := io.laststring.duplicate;
```

```
      io.putstring("\nEnter the third part of the triple : ");
      io.readline;
      text3 := io.laststring.duplicate;
      io.putstring("\nDo u wish to add any more ?  : Yes(y)\No(n) ");
      io.readchar;

      htabl1.do_adding(text1,text2,text3);
      htabl2.do_adding(text1,text2,text3);
      htabl3.do_adding(text1,text2,text3);
      htabl4.do_adding(text1,text2,text3);
      htabl5.do_adding(text1,text2,text3);
      htabl6.do_adding(text1,text2,text3);
      htabl7.do_adding(text1,text2,text3);

      if io.lastchar='n' then quit:=true end;
      io.input.next_line
    end
 end;

do_delete is

  do
    from
       quit:=false;
    until
       quit=true
    loop
       io.putstring("\nEnter the first part of the triple to be deleted: ");
       io.readline;
       text1 := io.laststring.duplicate;
       io.putstring("\nEnter the second part of the triple to be deleted: ");
       io.readline;
       text2 := io.laststring.duplicate;
       io.putstring("\nEnter the third part of the triple to be deleted: ");
```

```
        io.readline;
        text3 := io.laststring.duplicate;
        io.putstring("\nDo u wish to delete any more ? : Yes(y)\No(n) ");
        io.readchar;
        htab11.do_deleting(text1,text2,text3);
        htab12.do_deleting(text1,text2,text3);
        htab13.do_deleting(text1,text2,text3);
        htab14.do_deleting(text1,text2,text3);
        htab15.do_deleting(text1,text2,text3);
        htab16.do_deleting(text1,text2,text3);
        htab17.do_deleting(text1,text2,text3);

        if io.lastchar='n' then quit:=true end;
        io.input.next_line;
      end;
    end;

do_retrieve is
  do
    from
       quit:=false;
    until
       quit=true
    loop
       io.putstring("\nEnter ? where values are not known");
       io.new_line;
       io.putstring("\nEnter the first part of the triple to be retrieved: ");
       io.readline;
       text1 := io.laststring.duplicate;
       io.putstring("\nEnter the second part of the triple to be retrieved: ");
       io.readline;
       text2 := io.laststring.duplicate;
       io.putstring("\nEnter the third part of the triple to be retrieved: ");
       io.readline;
```

```
      text3 := io.laststring.duplicate;

      if (not(text1.equal("?")) and not(text2.equal("?")) and
          not(text3.equal("?"))) then
        htabl7.do_retrieving(text1,text2,text3)
      elsif ((text1.equal("?")) and (text2.equal("?"))) then
        htabl3.do_retrieving(text1,text2,text3)
      elsif ((text2.equal("?")) and (text3.equal("?"))) then
        htabl1.do_retrieving(text1,text2,text3)
      elsif ((text1.equal("?")) and (text3.equal("?"))) then
        htabl2.do_retrieving(text1,text2,text3)
      elsif (text1.equal("?")) then
        htabl5.do_retrieving(text1,text2,text3)
      elsif (text2.equal("?")) then
        htabl6.do_retrieving(text1,text2,text3)
      elsif (text3.equal("?")) then
        htabl4.do_retrieving(text1,text2,text3)
      end;

      io.putstring("\nAny more combinations of triples to retrieve ? : (y)\(n) ");
      io.readchar;
      if io.lastchar='n' then quit:=true end;
      io.input.next_line;
    end;
  end;

create_tree_file is
 -- This file creates the initial tree file

 local
   tabl: ARRAY[BINARY_TREE[INTEGER]];
   free_blk_list: ARRAY[INTEGER];
   camera: ENVIRONMENT;
   tablename: FILE;
```

```
do
  camera.Create;
  tablename.Create("Tree.env");
  camera.set_file(tablename);
  camera.open;
  tabl.Create(1,10);
  free_blk_list.Create(1,0);
  camera.put(tabl,"table1");
  camera.put(tabl,"table2");
  camera.put(tabl,"table3");
  camera.put(tabl,"table4");
  camera.put(tabl,"table5");
  camera.put(tabl,"table6");
  camera.put(tabl,"table7");
  camera.put(free_blk_list,"free1");
  camera.put(free_blk_list,"free2");
  camera.put(free_blk_list,"free3");
  camera.put(free_blk_list,"free4");
  camera.put(free_blk_list,"free5");
  camera.put(free_blk_list,"free6");
  camera.put(free_blk_list,"free7");
  if not camera.ok then
      io.putstring("NOT RECORDED");
      io.new_line
  end;
  camera.close;
  camera.store;
  tablename.add_permission ("ugo","w");
end;

do_initialization is
  do
    create_tree_file;
    htabl1.startup;
```

```
    htabl2.startup;
    htabl3.startup;
    htabl4.startup;
    htabl5.startup;
    htabl6.startup;
    htabl7.startup;
  end;

create_tables is
  do
    htabl1.Create;
    htabl2.Create;
    htabl3.Create;
    htabl4.Create;
    htabl5.Create;
    htabl6.Create;
    htabl7.Create;
  end;

end --class INTERACTION
```

§ A.2 Class BASIC_TABLE

```
-- The basic table with the three functions:
-- 1> To add a triple
-- 2> To delete a triple
-- 3> To retrieve a triple


class BASIC_TABLE export
```

```
    initialize,add_triple,delete_triple,retrieve_triple

feature

    tabl: ARRAY[BINARY_TREE[INTEGER]];
    aTree: BINARY_TREE[INTEGER];
    free_blk_list: ARRAY[INTEGER];
    camera: ENVIRONMENT;
    tablename: FILE;

initialize (flname,tbl_name,env_key,free_key: STRING) is

  local
    i: INTEGER;

  do
    camera.Create;
    tablename.Create(flname);
    camera.set_file(tablename);
    camera.retrieve;
    if camera.ok then
      tabl?=camera.item(env_key);
      from
        i:=1;
      until
        i>10
      loop
         --Create and assign the basic blocks to each index in the table
        aTree.Create(assign_initial_block(tbl_name.to_c));
        tabl.put(aTree,i);
        i:=i+1;
      end;
      camera.force(tabl,env_key);
      if not camera.ok then
```

```
      io.putstring("NOT RECORDED");
      io.new_line
    end;
    camera.close;
    camera.store;
    tablename.add_permission ("ugo","w");
  end;
end;


add_triple(tbl_no: INTEGER; flname,env_key,free_key,
            tbl_name,tpl_key,text1,text2,text3: STRING) is
  local
    i,j,pos,blk_address,inserted,free_blk_used,free_count,level,newblk_address: INTEGER;
    check_string,check_substr: STRING;
    check_bit: CHARACTER;
    temp_list: ARRAY[INTEGER];

  do
    camera.Create;
    tablename.Create(flname);
    camera.set_file(tablename);
    camera.retrieve;
    if camera.ok then
      from
        camera.open;
        inserted:=0;
        level:=0;
        pos:=ascii_eql(tpl_key) mod 10;
        if pos=0 then
          pos:=10;
        end;
        tabl?=camera.item(env_key);
        free_blk_list?=camera.item(free_key);
```

```
  check_string.Create(15);
until
  inserted=1
loop
  from
    aTree:=tabl.item(pos);
  until
    aTree.item /= -1
  loop
    level:=level+1;
    check_bit:=get_bit(tpl_key,level);
    if check_bit='0' then
      aTree:=aTree.left_child
    end;
    if check_bit='1' then
      aTree:=aTree.right_child
    end;
  end;
  blk_address:=aTree.item;
  free_count:=free_blk_list.count;
  check_string:=record_triple(free_blk_list.to_c,free_count,text1.to_c,
                              text2.to_c,text3.to_c,tbl_name.to_c,
                              blk_address,level,tbl_no);
  check_substr:=check_string.substring(1,1);
  inserted:=check_substr.to_integer;
  check_substr:=check_string.substring(2,2);
  free_blk_used:=check_substr.to_integer;
  check_substr:=check_string.substring(3,check_string.count);
  newblk_address:=check_substr.to_integer;
  temp_list.Create(1,(free_count-free_blk_used));
  from
    i:=1;
    j:=free_blk_used+1;
  until
```

```
        i>(free_count-free_blk_used)
      loop
        temp_list.put(free_blk_list.item(j),i);
        i:=i+1;
        j:=j+1;
      end;
      free_blk_list.wipe_out;
      free_blk_list:=temp_list;
      if newblk_address /= -1 then
        aTree.put(-1);
        aTree.child_put_left(blk_address);
        aTree.child_put_right(newblk_address);
      end;
      aTree.child_start;
      tabl.force(aTree,pos);
    end;
    camera.force(tabl,env_key);
    camera.force(free_blk_list,free_key);
    if not camera.ok then
      io.putstring("NOT RECORDED");
      io.new_line;
    end;
    camera.close;
    camera.store;
    tablename.add_permission("ugo","w");
  end
 end;

delete_triple(tbl_no: INTEGER; flname,env_key,free_key,
              tbl_name,tpl_key,text1,text2,text3: STRING) is
  local
    pos,level,blk_address,check_address,side,free_blk_ctr,value: INTEGER;
    check_bit: CHARACTER;
  do
```

```
camera.Create;
level:=0;
tablename.Create(flname);
camera.set_file(tablename);
camera.retrieve;
if camera.ok then
  pos:=ascii_eql(tpl_key) mod 10;
  if pos=0 then
    pos:=10;
  end;
  tabl?=camera.item(env_key);
  free_blk_list?=camera.item(free_key);
  camera.open;
  side:=-1;
  from
    aTree:=tabl.item(pos)
  until
    aTree.item /= -1
  loop
    level:=level+1;
    check_bit:=get_bit(tpl_key,level);
    if check_bit='0' then
      aTree:=aTree.left_child;
      side:=0
    end;
    if check_bit='1' then
       aTree:=aTree.right_child;
       side:=1
    end;
  end;
  blk_address:=aTree.item;
  check_address:=del_triple(tbl_no,text1.to_c,text2.to_c,text3.to_c,
                            tbl_name.to_c,blk_address);
  if check_address /= -1 then
```

```
    if level /= 0 then
      free_blk_ctr:=free_blk_list.count;
      free_blk_ctr:=free_blk_ctr+1;
      free_blk_list.force(check_address,free_blk_ctr);
      aTree:=aTree.parent;
      if side=0 then
        aTree:=aTree.right_child;
        value:=aTree.item;
        aTree:=aTree.parent;
        aTree.remove_left_child;
        aTree.remove_right_child;
        aTree.put(value);
      end;
     if side=1 then
       aTree:=aTree.left_child;
       value:=aTree.item;
       aTree:=aTree.parent;
       aTree.remove_left_child;
       aTree.remove_right_child;
       aTree.put(value);
      end;
    end;
  end;
  tabl.put(aTree,pos);
  camera.force(tabl,env_key);
  camera.force(free_blk_list,free_key);
  if not camera.ok then
    io.putstring("NOT RECORDED");
    io.new_line;
  end;
  camera.close;
  camera.store;
  tablename.add_permission("ugo","w");
end;
```

```
  end;

retrieve_triple(tbl_no: INTEGER; flname,env_key,tbl_name,
                tpl_key,text1,text2,text3: STRING) is
  local
    pos,blk_address,inserted,level,newblk_address: INTEGER;
    check_bit: CHARACTER;
  do
    camera.Create;
    level:=0;
    tablename.Create(flname);
    camera.set_file(tablename);
    camera.retrieve;
    if camera.ok then
      pos:=ascii_eql(tpl_key) mod 10;
      if pos=0 then
        pos:=10;
      end;
      tabl?=camera.item(env_key);
      from
        aTree:=tabl.item(pos);
      until
        aTree.is_root=true
      loop
        aTree:=aTree.parent;
      end;
      from
      until
        aTree.item /=-1
      loop
        level:=level+1;
        check_bit:=get_bit(tpl_key,level);
        if check_bit='0' then
          aTree:=aTree.left_child;
```

```
        end;
        if check_bit='1' then
           aTree:=aTree.right_child;
        end;
      end;
      blk_address:=aTree.item;
      ret_triple(tbl_no,text1.to_c,text2.to_c,text3.to_c,
                 tbl_name.to_c,blk_address);
    end;
  end;

get_bit(Key:STRING,location: INTEGER): CHARACTER is

  local
    char:CHARACTER;
    char_bit_sequence:STRING;
    bit_position,position:INTEGER;

  do
   char_bit_sequence.Create(6);
   position:=location mod Key.count;
   if position=0 then
     bit_position:=location div Key.count;
     position:=Key.count;
   else
     bit_position:=(location div Key.count) + 1;
   end;
   char:=Key.item(position);
   inspect
     char
   when 'a','A' then
     char_bit_sequence:="010110"
   when 'b','B' then
     char_bit_sequence:="100100"
```

```
when 'c','C' then
  char_bit_sequence:="110101"
when 'd','D' then
  char_bit_sequence:="001001"
when 'e','E' then
  char_bit_sequence:="010101"
when 'f','F' then
  char_bit_sequence:="001100"
when 'g','G' then
  char_bit_sequence:="010100"
when 'h','H' then
  char_bit_sequence:="111000"
when 'i','I' then
  char_bit_sequence:="000111"
when 'j','J' then
  char_bit_sequence:="110111"
when 'k','K' then
  char_bit_sequence:="000101"
when 'l','L' then
  char_bit_sequence:="110110"
when 'm','M' then
  char_bit_sequence:="110011"
when 'n','N' then
  char_bit_sequence:="110001"
when 'o','O' then
  char_bit_sequence:="100101"
when 'p','P' then
  char_bit_sequence:="101011"
when 'q','Q' then
  char_bit_sequence:="011101"
when 'r','R' then
  char_bit_sequence:="001110"
when 's','S' then
  char_bit_sequence:="011100"
```

```
    when 't','T' then
      char_bit_sequence:="101001"
    when 'u','U' then
      char_bit_sequence:="001010"
    when 'v','V' then
      char_bit_sequence:="011011"
    when 'w','W' then
      char_bit_sequence:="010010"
    when 'x','X' then
      char_bit_sequence:="100010"
    when 'y','Y' then
      char_bit_sequence:="101101"
    when 'z','Z' then
      char_bit_sequence:="000110"
    end;
    Result:=char_bit_sequence.item(bit_position)
   end;

assign_initial_block(t_name: INTEGER): INTEGER is

   external
      assign_initial_block(t_name: INTEGER): INTEGER
      name "assign_initial_block" language "C"
   do
      Result:=assign_initial_block(t_name);
   end;

record_triple(free_blk_list,free_count,t1,t2,t3,
              t_name,blk_address,level,t_no: INTEGER): STRING is

   external
      record_triple(free_blk_list,free_count,t1,t2,t3,
                    t_name,blk_address,level,t_no: INTEGER): STRING
      name "record_triple" language "C"
```

```
   do
     Result:=record_triple(free_blk_list,free_count,t1,t2,t3,
                           t_name,blk_address,level,t_no);
   end;

del_triple(tbl_no,t1,t2,t3,t_name,blk_address: INTEGER): INTEGER is

   external
      del_triple(tbl_no,t1,t2,t3,t_name,blk_address: INTEGER): INTEGER
      name "del_triple" language "C"
   do
     Result:=del_triple(tbl_no,t1,t2,t3,t_name,blk_address);
   end;

ret_triple(tbl_no,t1,t2,t3,t_name,blk_address: INTEGER) is

   external
      ret_triple(tbl_no,t1,t2,t3,t_name,blk_address: INTEGER)
      name "ret_triple" language "C"
   do
     ret_triple(tbl_no,t1,t2,t3,t_name,blk_address);
   end;


ascii_eql(text: STRING): INTEGER is

   local
      temp:INTEGER;
      temp1:BASIC_ROUT;

   do
    temp1.Create;
    from
      temp:=1;
```

```
    until
      temp=text.count+1
    loop
      Result:=Result+(temp1.charcode(text.item(temp)));
      temp:=temp+1;
    end;
   end;

end -- BASIC_TABLE
```

```
/* FILE : types_decl.c */
#include <fcntl.h>
#include <ctype.h>
#include "/usr/local/ucc/eiffel/Eiffel/files/_eiffel.h"
extern OBJPTR eif_create();
extern ROUT_PTR eif_rout();

typedef struct triple {
  char text1[50];
  char text2[50];
  char text3[50];
  int next;
 } TRIPLE;

typedef struct block {
  int header;
  TRIPLE record[5];
  int next;
 } BLOCK;
```

```
/* FILE : rec_del_ret.c */
/* This module is the C interface of the Eiffel program. It has three functions
   1. For storing the triple
   2. For deleting a triple
   3. For retrieving a triple  */

#include <fcntl.h>
#include <ctype.h>
#include "/usr/local/ucc/eiffel/Eiffel/files/_eiffel.h"
extern OBJPTR eif_create();
extern ROUT_PTR eif_rout();

typedef struct triple {
  char text1[50];
  char text2[50];
  char text3[50];
  int next;
 } TRIPLE;

typedef struct block {
  int header;
  TRIPLE record[5];
  int next;
 } BLOCK;

TRIPLE tple;
BLOCK blk,main_blk,left_blk,right_blk,next_blk;
int i,count,flag=0,temp,ret_address,prev,first,last;
int left=1,right=1,inserted,main_count,checked,ct;
int free_blk,fdesc,found,no_alloted,total_free_blks,update;

/* This function stores the triple in the appropriate block */

OBJPTR record_triple(int *free_list,int total_free_blks,char *t1,char *t2,
```

```
                    char *t3,char *tbl_name,int blk_address,int level,int t_no)

{
  OBJPTR obj_string;
  ROUT_PTR to_c, string_append;
  char ret[6],*st;


  obj_string = eif_create ("string", 6);
  string_append = eif_rout (obj_string, "append");
  fdesc=open(tbl_name,O_RDWR);
  if (fdesc == -1) {
    printf("\n ERROR : Cannot open tabl.dat\n");
    exit(1);
  }
  lseek(fdesc,0L,0);
  lseek(fdesc,blk_address,0);
  read(fdesc,&blk,sizeof(BLOCK));
  for(i=0;i<blk.header;i++)
    printf(" %d %s %s %s %d %d\n",blk.header,blk.record[i].text1, blk.record[i].text2,
            blk.record[i].text3,blk.record[i].next,blk_address);
  main_count=blk.header;
  ret_address=-1;flag=0;inserted=0;no_alloted=0;
  update=0;last=0;checked=0;first=1;

  /* check if main block is 0 */

  if (main_count > 0)
   {
     flag=0;count=0;checked=0;
     while ((count<main_count) && (!checked))
     {
       switch (t_no)
       {
```

```
 case 1 : if (strcmp(blk.record[count].text1,t1)==0)
            checked=1;
          break;
 case 2 : if (strcmp(blk.record[count].text2,t2)==0)
            checked=1;
          break;
 case 3 : if (strcmp(blk.record[count].text3,t3)==0)
             checked=1;
          break;
 case 4 : if ((strcmp(blk.record[count].text1,t1)==0) &&
             (strcmp(blk.record[count].text2,t2)==0))
             checked=1;
          break;
 case 5 : if ((strcmp(blk.record[count].text2,t2)==0) &&
              (strcmp(blk.record[count].text3,t3)==0))
              checked=1;
          break;
 case 6 : if ((strcmp(blk.record[count].text1,t1)==0) &&
             (strcmp(blk.record[count].text3,t3)==0))
              checked=1;
          break;
 case 7 : if ((strcmp(blk.record[count].text1,t1)==0) &&
              (strcmp(blk.record[count].text2,t2)==0) &&
              (strcmp(blk.record[count].text3,t3)==0))
              checked=1;
          break;
}
if (checked)
{
  flag=1;
  temp=blk.record[count].next;
  printf("Temp = %d \n",temp);
  while (temp != -1)
  {
```

```
 fgets(st,sizeof(char),stdin);
 scanf("%s",st);
 first=0;
 lseek(fdesc,0L,0);
 lseek(fdesc,temp,0);
 read(fdesc,&blk,sizeof(BLOCK));
 prev=temp;
 temp=blk.next;
 printf("TEMP = %d \n",temp);
}

if (first)
{
  if ((total_free_blks != 0) && (no_alloted != total_free_blks))
   {
     free_blk=free_list[no_alloted];
     no_alloted++;
   }
   else
   {
     lseek(fdesc,0L,0);
     read(fdesc,&free_blk,4);
     update=1;
   }
  blk.record[count].next=free_blk;
  lseek(fdesc,0L,0);
  lseek(fdesc,blk_address,0);
  write(fdesc,(char *) &blk,sizeof(BLOCK));
  do_rest(t1,t2,t3);
}

if ((blk.header == 5) &&  !first)
{
  last=1;
```

```
 if ((total_free_blks != 0) && (no_alloted != total_free_blks))
  {
    free_blk=free_list[no_alloted];
    no_alloted++;
   }
  else
  {
    lseek(fdesc,0L,0);
    read(fdesc,&free_blk,4);
    update=1;
  }
 printf("Free block assigned = %d\n",free_blk);
 blk.next=free_blk;

 for (ct=0;ct<blk.header;ct++)
   printf(" %s %s %s %d \n",blk.record[ct].text1,blk.record[ct].text2,
            blk.record[ct].text3,blk.next);
 lseek(fdesc,0L,0);
 lseek(fdesc,prev,0);
 write(fdesc,(char *) &blk,sizeof(BLOCK));
 do_rest(t1,t2,t3);
}

if ((blk.header < 5) && !first && !last)
{
  printf("In if ((blk.header < 5) && !first && !last) \n");
  for(ct=0;ct<blk.header;ct++)
  printf("%d %d %s  %s  %s %d\n",prev,blk.header,blk.record[ct].text1,
          blk.record[ct].text2,blk.record[ct].text3,blk.next);
  printf("END\n");
  blk.header=blk.header+1;
  strcpy(blk.record[blk.header-1].text1,t1);
  strcpy(blk.record[blk.header-1].text2,t2);
  strcpy(blk.record[blk.header-1].text3,t3);
```

```
        for(ct=0;ct<blk.header;ct++)
        printf("%d %d %s  %s  %s %d\n",prev,blk.header,blk.record[ct].text1,
                blk.record[ct].text2,blk.record[ct].text3,blk.next);
        lseek(fdesc,0L,0);
        lseek(fdesc,prev,0);
        write(fdesc,(char *) &blk,sizeof(BLOCK));
        inserted=1;
      }

    }
   count++;
  }

  /* check if main block is 5 */
if ((main_count==5) && (!inserted))
 {
   flag=1;i=0;left=0;right=0;
   level++;
   while (i<5)
   {
     if (get_bitC(blk.record[i].text1,level)=='0')
     {
       strcpy(left_blk.record[left].text1,blk.record[i].text1);
       strcpy(left_blk.record[left].text2,blk.record[i].text2);
       strcpy(left_blk.record[left].text3,blk.record[i].text3);
       left_blk.record[left].next=blk.record[i].next;
       left++;
     }
     if (get_bitC(blk.record[i].text1,level)=='1')
     {
       strcpy(right_blk.record[right].text1,blk.record[i].text1);
       strcpy(right_blk.record[right].text2,blk.record[i].text2);
       strcpy(right_blk.record[right].text3,blk.record[i].text3);
       right_blk.record[right].next=blk.record[i].next;
```

```
    right++;
  }
  i++;
}

if ((left==5) || (right==5))
  inserted=0;
left_blk.header=left;
left_blk.next=-1;
lseek(fdesc,0L,0);
lseek(fdesc,blk_address,0);
write(fdesc,(char *) &left_blk,sizeof(BLOCK));
right_blk.header=right;
right_blk.next=-1;
if ((total_free_blks != 0) && (no_alloted != total_free_blks))
{
  free_blk=free_list[no_alloted];
  no_alloted++;
}
else
{
  lseek(fdesc,0L,0);
  read(fdesc,&free_blk,4);
  update=1;
}
ret_address=free_blk;
lseek(fdesc,0L,0);
lseek(fdesc,free_blk,0);
write(fdesc,(char *) &right_blk,sizeof(BLOCK));
if (update)
{
  lseek(fdesc,0L,0);
  free_blk=lseek(fdesc,free_blk+sizeof(BLOCK),0);
  lseek(fdesc,0L,0);
```

```
      write(fdesc,(char *) &free_blk,4);
    }
  }
 }
 if ((main_count == 0) || ((main_count<5) && (flag==0)))
  {
    blk.header=blk.header+1;
    strcpy(blk.record[blk.header-1].text1,t1);
    strcpy(blk.record[blk.header-1].text2,t2);
    strcpy(blk.record[blk.header-1].text3,t3);
    blk.record[blk.header-1].next = -1;
    lseek(fdesc,blk_address,0);
    write(fdesc,&blk,sizeof(BLOCK));
    inserted=1;
  }
 close(fdesc);
 sprintf(ret,"%d%d%d",inserted,no_alloted,ret_address);
 (*string_append) (obj_string, MakeStr (ret));
 to_c=eif_rout(obj_string,"to_c");
 return cbj_string;
}

do_rest (char *t1,char *t2,char *t3)
 {
   strcpy(blk.record[0].text1,t1);
   strcpy(blk.record[0].text2,t2);
   strcpy(blk.record[0].text3,t3);
   blk.header=1;
   blk.next= -1;
   lseek(fdesc,0L,0);
   lseek(fdesc,free_blk,0);
   write(fdesc,(char *) &blk,sizeof(BLOCK));

   if (update)
```

```
    {
      update=0;
      lseek(fdesc,0L,0);
      free_blk=lseek(fdesc,free_blk+sizeof(BLOCK),0);

      lseek(fdesc,0L,0);
      write(fdesc,(char *) &free_blk,4);
    }
    inserted=1;
  }


char get_bitC (char *key,int location)
{
  char *char_bit_sequence;
  int  bit_position,position;

  position=location%(strlen(key));
  if (position==0)
  {
    bit_position=location/(strlen(key));
    position=strlen(key);
  }
  else
    bit_position=(location/strlen(key))+1;


  switch (toupper(key[position-1]))
  {
   case 'A' : char_bit_sequence="010110";
              break;
   case 'B' : char_bit_sequence="100100";
              break;
   case 'C' : char_bit_sequence="110101";
```

```
            break;
case 'D' : char_bit_sequence="001001";
            break;
case 'E' : char_bit_sequence="010101";
            break;
case 'F' : char_bit_sequence="001100";
            break;
case 'G' : char_bit_sequence="010100";
            break;
case 'H' : char_bit_sequence="111000";
            break;
case 'I' : char_bit_sequence="000111";
            break;
case 'J' : char_bit_sequence="110111";
            break;
case 'K' : char_bit_sequence="000101";
            break;
case 'L' : char_bit_sequence="110110";
            break;
case 'M' : char_bit_sequence="110011";
            break;
case 'N' : char_bit_sequence="110001";
            break;
case 'O' : char_bit_sequence="100101";
            break;
case 'P' : char_bit_sequence="101011";
            break;
case 'Q' : char_bit_sequence="011101";
            break;
case 'R' : char_bit_sequence="001110";
            break;
case 'S' : char_bit_sequence="011100";
            break;
case 'T' : char_bit_sequence="101001";
```

```
                break;
    case 'U' : char_bit_sequence="001010";
                break;
    case 'V' : char_bit_sequence="011011";
                break;
    case 'W' : char_bit_sequence="010010";
                break;
    case 'X' : char_bit_sequence="100010";
                break;
    case 'Y' : char_bit_sequence="101101";
                break;
    case 'Z' : char_bit_sequence="000110";
                break;
   }
   return (char_bit_sequence[bit_position-1]);
}

/* This function deletes the triple from the block */

int del_triple(int tbl_no,char *t1,char *t2, char *t3,
                char *tbl_name,int blk_address)
{
   int k,next_blk_address,depth;
   BLOCK next_blk;

   fdesc=open(tbl_name,O_RDWR);
   if (fdesc == -1) {
     printf("\n ERROR : Cannot open tabl%d.dat\n",tbl_no);
     exit(1);
   }
   lseek(fdesc,0L,0);
   lseek(fdesc,blk_address,0);
   read(fdesc,&blk,sizeof(BLOCK));
   main_count=blk.header;
```

```
ret_address=-1;

if (main_count > 0)
{
  count=0;found=0;
  switch (tbl_no)
   {
    case 1: while (!found)
            {
             if ((strcmp(blk.record[count].text1,t1)==0))
             {
               if ((strcmp(blk.record[count].text2,t2)==0) &&
                    (strcmp(blk.record[count].text3,t3)==0))
                {
                  found=1;
                  do_deletion(count,blk,blk_address);
                }
               else
                {
                  next_blk_address=blk.record[count].next; depth=1;
                  while (next_blk_address != -1)
                  {
                    lseek(fdesc,0L,0);          /* rewind the file */
                    lseek(fdesc,next_blk_address,0);
                    read(fdesc,&next_blk,sizeof(BLOCK));
                    k=0;
                    while (k<next_blk.header)
                    {
                     if ((strcmp(next_blk.record[k].text2,t2)==0) &&
                         (strcmp(next_blk.record[k].text3,t3)==0))
                     {
                       found=1;
                       do_other_deletion(k,next_blk,next_blk_address,count,
                                         blk,blk_address,depth);
```

```
                printf("TRIPLE DELETED\n");
              }
              k++;
              if (k==next_blk.header)
              {
                blk_address=next_blk_address;
                next_blk_address=next_blk.next;
                depth++;
              }
             }
            }
          }
         }
         count++;
         if ((count==blk.header) && (!found))
          {
            found=1;
            printf("This triple does not exist");
          }
        }
        break;

case 2: while (!found)
        {
         if ((strcmp(blk.record[count].text2,t2)==0))
         {
           if ((strcmp(blk.record[count].text1,t1)==0) &&
               (strcmp(blk.record[count].text3,t3)==0))
            {
              found=1;
              do_deletion(count,blk,blk_address);
            }
           else
            {
```

```
    next_blk_address=blk.record[count].next; depth=1;
    while (next_blk_address != -1)
    {
      lseek(fdesc,0L,0);          /* rewind the file */
      lseek(fdesc,next_blk_address,0);
      read(fdesc,&next_blk,sizeof(BLOCK));
      k=0;
      while (k<next_blk.header)
      {
       if ((strcmp(next_blk.record[k].text2,t2)==0) &&
           (strcmp(next_blk.record[k].text3,t3)==0))
       {
         found=1;
         do_other_deletion(k,next_blk,next_blk_address,count,
                           blk,blk_address,depth);
       }
       k++;
       if (k==next_blk.header)
       {
         blk_address=next_blk_address;
         next_blk_address=next_blk.next;
         depth++;
       }
      }
     }
   }
}
count++;
if ((count==blk.header) && (!found))
 {
   found=1;
   printf("This triple does not exist");
 }
}
```

```
        break;

case 3: while (!found)
        {
         if ((strcmp(blk.record[count].text3,t3)==0))
         {
           if ((strcmp(blk.record[count].text1,t1)==0) &&
               (strcmp(blk.record[count].text2,t2)==0))
            {
              found=1;
              do_deletion(count,blk,blk_address);
            }
           else
            {
              next_blk_address=blk.record[count].next; depth=1;
              while (next_blk_address != -1)
              {
                lseek(fdesc,0L,0);          /* rewind the file */
                lseek(fdesc,next_blk_address,0);
                read(fdesc,&next_blk,sizeof(BLOCK));
                k=0;
                while (k<next_blk.header)
                {
                 if ((strcmp(next_blk.record[k].text2,t2)==0) &&
                     (strcmp(next_blk.record[k].text3,t3)==0))
                 {
                   found=1;
                   do_other_deletion(k,next_blk,next_blk_address,count,
                                     blk,blk_address,depth);
                 }
                 k++;
                 if (k==next_blk.header)
                 {
                   blk_address=next_blk_address;
```

```
                next_blk_address=next_blk.next;
                depth++;
              }
             }
            }
           }
         }
         count++;
         if ((count==blk.header) && (!found))
          {
            found=1;
            printf("This triple does not exist");
          }
        }
        break;

case 4: while (!found)
        {
         if ((strcmp(blk.record[count].text1,t1)==0) &&
             (strcmp(blk.record[count].text2,t2)==0))
         {
           if (strcmp(blk.record[count].text3,t3)==0)
            {
              found=1;
              do_deletion(count,blk,blk_address);
            }
           else
            {
              next_blk_address=blk.record[count].next; depth=1;
              while (next_blk_address != -1)
              {
                lseek(fdesc,0L,0);          /* rewind the file */
                lseek(fdesc,next_blk_address,0);
                read(fdesc,&next_blk,sizeof(BLOCK));
```

```
                k=0;
                while (k<next_blk.header)
                {
                 if ((strcmp(next_blk.record[k].text2,t2)==0) &&
                     (strcmp(next_blk.record[k].text3,t3)==0))
                 {
                   do_other_deletion(k,next_blk,next_blk_address,count,
                                     blk,blk_address,depth);
                   found=1;
                 }
                 k++;
                 if (k==next_blk.header)
                 {
                   blk_address=next_blk_address;
                   next_blk_address=next_blk.next;
                   depth++;
                 }
                }
              }
            }
           }
           count++;
           if ((count==blk.header) && (!found))
            {
              found=1;
              printf("This triple does not exist");
            }
          }
          break;
case 5: while (!found)
          {
           if ((strcmp(blk.record[count].text2,t2)==0) &&
               (strcmp(blk.record[count].text3,t3)==0))
           {
```

```
if (strcmp(blk.record[count].text1,t1)==0)
 {
   found=1;
   do_deletion(count,blk,blk_address);
 }
else
 {
   next_blk_address=blk.record[count].next; depth=1;
   while (next_blk_address != -1)
   {
     lseek(fdesc,0L,0);          /* rewind the file */
     lseek(fdesc,next_blk_address,0);
     read(fdesc,&next_blk,sizeof(BLOCK));
     k=0;
     while (k<next_blk.header)
     {
      if ((strcmp(next_blk.record[k].text2,t2)==0) &&
          (strcmp(next_blk.record[k].text3,t3)==0))
      {
        found=1;
        do_other_deletion(k,next_blk,next_blk_address,
                          count,blk,blk_address,depth);
      }
      k++;
      if (k==next_blk.header)
      {
         blk_address=next_blk_address;
         next_blk_address=next_blk.next;
         depth++;
      }
     }
   }
  }
}
```

```
        count++;
        if ((count==blk.header) && (!found))
         {
           found=1;
           printf("This triple does not exist");
         }
       }
       break;

case 6: while (!found)
       {
        if ((strcmp(blk.record[count].text1,t1)==0) &&
            (strcmp(blk.record[count].text3,t3)==0))
        {
          if (strcmp(blk.record[count].text2,t2)==0)
           {
             found=1;
             do_deletion(count,blk,blk_address);
           }
          else
           {
             next_blk_address=blk.record[count].next; depth=1;
             while (next_blk_address != -1)
             {
               lseek(fdesc,0L,0);          /* rewind the file */
               lseek(fdesc,next_blk_address,0);
               read(fdesc,&next_blk,sizeof(BLOCK));
               k=0;
               while (k<next_blk.header)
               {
                if ((strcmp(next_blk.record[k].text2,t2)==0) &&
                    (strcmp(next_blk.record[k].text3,t3)==0))
                {
                  found=1;
```

```
                    do_other_deletion(k,next_blk,next_blk_address,count,
                                      blk,blk_address,depth);
                   }
                  k++;
                  if (k==next_blk.header)
                   {
                     blk_address=next_blk_address;
                     next_blk_address=next_blk.next;
                     depth++;
                   }
                 }
               }
             }
            count++;
            if ((count==blk.header) && (!found))
             {
               found=1;
               printf("This triple does not exist");
             }
           }
           break;

case 7: while (!found)
           {
            if ((strcmp(blk.record[count].text3,t3)==0) &&
                (strcmp(blk.record[count].text1,t1)==0) &&
                (strcmp(blk.record[count].text2,t2)==0))
            {
              found=1;
              do_deletion(count,blk,blk_address);
            }
            else
            {
```

```
next_blk_address=blk.record[count].next; depth=1;
while (next_blk_address != -1)
{
  lseek(fdesc,0L,0);          /* rewind the file */
  lseek(fdesc,next_blk_address,0);
  read(fdesc,&next_blk,sizeof(BLOCK));
  k=0;
  while (k<next_blk.header)
  {
    if ((strcmp(next_blk.record[k].text2,t2)==0) &&
        (strcmp(next_blk.record[k].text3,t3)==0))
    {
      found=1;
      do_other_deletion(k,next_blk,next_blk_address,count,
                        blk,blk_address,depth);
    }
    k++;
    if (k==next_blk.header)
     {
       blk_address=next_blk_address;
       next_blk_address=next_blk.next;
       depth++;
     }
   }
  }
}
count++;
if ((count==blk.header) && (!found))
 {
   found=1;
   printf("This triple does not exist");
 }
}
break;
```

```
    }
  }

  if (main_count==0)
    printf("This triple does not exist\n");

  return (ret_address);
}

do_deletion(int count,BLOCK blk,int blk_address)
{
  int j,free,next_blk_address,depth;


  /* If the main block contains just this triple and no chains */
  free=0;
  if (blk.record[count].next == -1)
  {
   if (count!=(blk.header-1))
   {
    for(j=count+1; j<=main_count; j++,count++)
     {
       strcpy(blk.record[count].text1,blk.record[j].text1);
       strcpy(blk.record[count].text2,blk.record[j].text2);
       strcpy(blk.record[count].text3,blk.record[j].text3);
       blk.record[count].next=blk.record[j].next;
     }
   }
       blk.header=blk.header-1;
       lseek(fdesc,0L,0);
       lseek(fdesc,blk_address,0);
       write(fdesc,(char *) &blk,sizeof(BLOCK));
   /*    if (blk.header==0)
         ret_address=blk_address; */
```

```
}
else

/* If the main block contains chains */

{
  next_blk_address=blk.record[count].next;depth=1;
  while (next_blk_address != -1)
  {
    lseek(fdesc,0L,0);
    lseek(fdesc,next_blk_address,0);
    read(fdesc,&next_blk,sizeof(BLOCK));
    strcpy(blk.record[count].text1,next_blk.record[0].text1);
    strcpy(blk.record[count].text2,next_blk.record[0].text2);
    strcpy(blk.record[count].text3,next_blk.record[0].text3);
    if (next_blk.header==1)
    {
      blk.record[count].next=-1;
      ret_address=next_blk_address;
    }
    lseek(fdesc,0L,0);
    lseek(fdesc,blk_address,0);
    write(fdesc,(char *) &blk,sizeof(BLOCK));
    if (next_blk.header != 1)
    {
      /* storing in a temporary block blk */

      for(count=0,j=count+1; j<next_blk.header; count++,j++)
      {
        strcpy(blk.record[count].text1,next_blk.record[j].text1);
        strcpy(blk.record[count].text2,next_blk.record[j].text2);
        strcpy(blk.record[count].text3,next_blk.record[j].text3);
      }
      if (next_blk.next==-1)
```

```
          blk.next=-1;
        free=0;
        blk_address=next_blk_address;
      }
      else
      {
        ret_address=next_blk_address;
        free=1;
      }
      next_blk_address=next_blk.next;
    }
    if (free)
      blk.next=-1;
    blk.header=count;
    for(i=0;i<blk.header;i++)
        printf("%s %s %s %d\n",blk.record[i].text1,blk.record[i].text2,
                blk.record[i].text3,blk.next);
    lseek(fdesc,0L,0);
    lseek(fdesc,blk_address,0);
    write(fdesc,(char *) &blk,sizeof(BLOCK));
  }
}

do_other_deletion(int k,BLOCK blk,int blk_address,int old,
                  BLOCK old_blk,int old_blk_address,int depth)
{
 int i,j,l,m,next_blk_address,free,in_while;

 free=0;
 if (blk.next==-1)
 {
   if (k!=blk.header)
   {
     for (l=k,m=k+1;m<blk.header;l++,m++)
```

```
    {
     strcpy(blk.record[l].text1,blk.record[m].text1);
     strcpy(blk.record[l].text2,blk.record[m].text2);
     strcpy(blk.record[l].text3,blk.record[m].text3);
    }
  }
  blk.header=blk.header-1;
  lseek(fdesc,0L,0);
  lseek(fdesc,blk_address,0);
  write(fdesc,(char *) &blk,sizeof(BLOCK));
  ret_address=-1;
  if (blk.header==0)
  {
    ret_address=blk_address;
    if (depth==1)
    {
      old_blk.record[old].next=-1;
      lseek(fdesc,0L,0);
      lseek(fdesc,old_blk_address,0);
      write(fdesc,(char *) &old_blk,sizeof(BLOCK));
    }
    else
    {
      old_blk.next=-1;
      lseek(fdesc,0L,0);
      lseek(fdesc,old_blk_address,0);
      write(fdesc,(char *) &old_blk,sizeof(BLOCK));
    }
  }
 }
else
{
 next_blk_address=blk.next;
 for (i=k,m=k+1;m<blk.header;i++,m++)
```

```
 {
   printf("In for (i=k,m=k+1;m<=blk.header;i++,m++)\n");
   strcpy(blk.record[i].text1,blk.record[m].text1);
   strcpy(blk.record[i].text2,blk.record[m].text2);
   strcpy(blk.record[i].text3,blk.record[m].text3);
 }
while (next_blk_address != -1)
 {
   lseek(fdesc,0L,0);
   lseek(fdesc,next_blk_address,0);
   read(fdesc,&next_blk,sizeof(BLOCK));
   strcpy(blk.record[i].text1,next_blk.record[0].text1);
   strcpy(blk.record[i].text2,next_blk.record[0].text2);
   strcpy(blk.record[i].text3,next_blk.record[0].text3);
   lseek(fdesc,0L,0);
   lseek(fdesc,blk_address,0);
   write(fdesc,(char *) &blk,sizeof(BLOCK));
   if (next_blk.header != 1)
   {
     /* storing in a temporary block blk */

     for(i=0,j=i+1; j<next_blk.header; i++,j++)
     {
       strcpy(blk.record[i].text1,next_blk.record[j].text1);
       strcpy(blk.record[i].text2,next_blk.record[j].text2);
       strcpy(blk.record[i].text3,next_blk.record[j].text3);
     }
     free=0;
     blk_address=next_blk_address;
   }
   else
   {
     ret_address=next_blk_address;
     free=1;
```

```
    }
    next_blk_address=next_blk.next;
  }
  if (free)
   blk.next=-1;
  blk.header=i;
  lseek(fdesc,0L,0);
  lseek(fdesc,blk_address,0);
  write(fdesc,(char *) &blk,sizeof(BLOCK));
 }
}

/* This function retrieves the triple from the block */

ret_triple(int tbl_no,char *t1,char *t2, char *t3,char *tbl_name,int blk_address)
{
  int next_blk_address;

  fdesc=open(tbl_name,O_RDWR);
  if (fdesc == -1) {
    printf("\n ERROR : Cannot open tabl%d.dat\n",tbl_no);
    exit(1);
  }
  lseek(fdesc,0L,0);
  lseek(fdesc,blk_address,0);
  read(fdesc,&blk,sizeof(BLOCK));
  for (i=0;i<blk.header;i++)
    printf(" %s %s %s %d \n",blk.record[i].text1,blk.record[i].text2,
            blk.record[i].text3,blk.record[i].next);
  main_count=blk.header;

  if (main_count > 0)
  {
    count=0;found=0;
```

```
switch (tbl_no)
 {
  case 1: while (!found)
          {
           if ((strcmp(blk.record[count].text1,t1)==0))
           {
             printf("\n %s  %s  %s\n",blk.record[count].text1,
                    blk.record[count].text2,blk.record[count].text3);
             found=1; next_blk_address=blk.record[count].next;
             while (next_blk_address != -1)
             {
               lseek(fdesc,0L,0);          /* rewind the file */
               lseek(fdesc,next_blk_address,0);
               read(fdesc,&blk,sizeof(BLOCK));
               for(i=0;i<blk.header;i++)
                 printf("\n %s  %s  %s\n",blk.record[i].text1,
                        blk.record[i].text2,blk.record[i].text3);
               next_blk_address=blk.next;
             }
            }
           count++;
           if ((count==blk.header) && (!found))
            {
              found=1;
              printf("This triple does not exist");
            }
          }
          break;

   case 2: while (!found)
           {
            if ((strcmp(blk.record[count].text2,t2)==0))
            {
              printf("\n %s  %s  %s\n",blk.record[count].text1,
```

```
                blk.record[count].text2,blk.record[count].text3);
          found=1; next_blk_address=blk.record[count].next;
          while (next_blk_address != -1)
          {
            lseek(fdesc,0L,0);          /* rewind the file */
            lseek(fdesc,next_blk_address,0);
            read(fdesc,&blk,sizeof(BLOCK));
            for(i=0;i<blk.header;i++)
              printf("\n %s  %s  %s\n",blk.record[i].text1,
                     blk.record[i].text2,blk.record[i].text3);
            next_blk_address=blk.next;
          }
         }
         count++;
         if ((count==blk.header) && (!found))
          {
            found=1;
            printf("This triple does not exist");
          }
        }
        break;

case 3: while (!found)
        {
         if ((strcmp(blk.record[count].text3,t3)==0))
         {
           printf("\n %s  %s  %s\n",blk.record[count].text1,
                  blk.record[count].text2,blk.record[count].text3);
           found=1; next_blk_address=blk.record[count].next;
           while (next_blk_address != -1)
           {
             lseek(fdesc,0L,0);          /* rewind the file */
             lseek(fdesc,next_blk_address,0);
             read(fdesc,&blk,sizeof(BLOCK));
```

```
            for(i=0;i<blk.header;i++)
              printf("\n %s  %s  %s\n",blk.record[i].text1,
                     blk.record[i].text2,blk.record[i].text3);
            next_blk_address=blk.next;
           }
          }
         count++;
         if ((count==blk.header) && (!found))
          {
            found=1;
            printf("This triple does not exist");
          }
        }
        break;

case 4: while (!found)
         {
          if ((strcmp(blk.record[count].text1,t1)==0) &&
              (strcmp(blk.record[count].text2,t2)==0))
          {
            printf("\n %s  %s  %s\n",blk.record[count].text1,
                   blk.record[count].text2,blk.record[count].text3);
            found=1; next_blk_address=blk.record[count].next;
            while (next_blk_address != -1)
            {
              lseek(fdesc,0L,0);          /* rewind the file */
              lseek(fdesc,next_blk_address,0);
              read(fdesc,&blk,sizeof(BLOCK));
              for(i=0;i<blk.header;i++)
                printf("\n %s  %s  %s\n",blk.record[i].text1,
                       blk.record[i].text2,blk.record[i].text3);
              next_blk_address=blk.next;
             }
            }
```

```
            count++;
            if ((count==blk.header) && (!found))
             {
               found=1;
               printf("This triple does not exist");
             }
           }
           break;

case 5: while (!found)
           {
            if ((strcmp(blk.record[count].text2,t2)==0) &&
                (strcmp(blk.record[count].text3,t3)==0))
            {
              printf("\n %s  %s  %s\n",blk.record[count].text1,
                     blk.record[count].text2,blk.record[count].text3);
              found=1; next_blk_address=blk.record[count].next;
              while (next_blk_address != -1)
              {
                lseek(fdesc,0L,0);          /* rewind the file */
                lseek(fdesc,next_blk_address,0);
                read(fdesc,&blk,sizeof(BLOCK));
                for(i=0;i<blk.header;i++)
                  printf("\n %s  %s  %s\n",blk.record[i].text1,
                         blk.record[i].text2,blk.record[i].text3);
                next_blk_address=blk.next;
              }
             }
            count++;
            if ((count==blk.header) && (!found))
             {
               found=1;
               printf("This triple does not exist");
             }
```

```
        }
        break;

case 6: while (!found)
        {
         if ((strcmp(blk.record[count].text1,t1)==0) &&
             (strcmp(blk.record[count].text3,t3)==0))
         {
           printf("\n %s  %s  %s\n",blk.record[count].text1,
                  blk.record[count].text2,blk.record[count].text3);
           found=1; next_blk_address=blk.record[count].next;
           while (next_blk_address != -1)
           {
             lseek(fdesc,0L,0);          /* rewind the file */
             lseek(fdesc,next_blk_address,0);
             read(fdesc,&blk,sizeof(BLOCK));
             for(i=0;i<blk.header;i++)
               printf("\n %s  %s  %s\n",blk.record[i].text1,
                      blk.record[i].text2,blk.record[i].text3);
             next_blk_address=blk.next;
           }
         }
         count++;
         if ((count==blk.header) && (!found))
          {
            found=1;
            printf("This triple does not exist");
          }
        }
        break;

case 7: while (!found)
        {
         if ((strcmp(blk.record[count].text1,t1)==0) &&
```

```
                (strcmp(blk.record[count].text2,t2)==0) &&
                (strcmp(blk.record[count].text3,t3)==0))
              {
                printf("\n %s  %s  %s\n",blk.record[count].text1,
                       blk.record[count].text2,blk.record[count].text3);
                found=1; next_blk_address=blk.record[count].next;
                while (next_blk_address != -1)
                {
                  lseek(fdesc,0L,0);          /* rewind the file */
                  lseek(fdesc,next_blk_address,0);
                  read(fdesc,&blk,sizeof(BLOCK));
                  for(i=0;i<blk.header;i++)
                    printf("\n %s  %s  %s\n",blk.record[i].text1,
                           blk.record[i].text2,blk.record[i].text3);
                  next_blk_address=blk.next;
                }
               }
              count++;
              if ((count==blk.header) && (!found))
               {
                 found=1;
                 printf("This triple does not exist");
               }
             }
             break;
        }
    }
    if (main_count==0)
    {
      printf("This triple does not exist");
    }
}
```

```
/* FILE: assign_initial_block.c */
#include <fcntl.h>

int assign_initial_block(char *tbl_name)
{
  typedef struct triple {
    char text1[50];
    char text2[50];
    char text3[50];
    int next;
  } TRIPLE;

  typedef struct block {
  int header;
  TRIPLE record[5];
  int next;
  } BLOCK;

  BLOCK blk;

  int free_blk;
  int start_blk,offset,fdesc;

  fdesc=open(tbl_name,O_RDWR | O_CREAT, 0666);
  if (fdesc == -1) {
    printf("\n ERROR : Cannot open %s \n",tbl_name);
    exit(1);
  }
  offset=lseek(fdesc,sizeof(int),0);
  lseek(fdesc,0L,0);
  write(fdesc,(char *) &offset,4);
  lseek(fdesc,0L,0);
  read(fdesc,&offset,4);
```

```
 start_blk = offset;
 blk.header = 0;
 blk.next = -1;
 lseek(fdesc,offset,0);
 write(fdesc,(char *) &blk,sizeof(BLOCK));
 lseek(fdesc,0L,0);
 free_blk = lseek(fdesc,offset+sizeof(BLOCK),0);
 lseek(fdesc,0L,0);
 write(fdesc,(char *) &free_blk,4);
 close(fdesc);
 return(start_blk);
}
```

§ A.3 Class HASH_TABLE1

```
    -- The hash table that maps on the first part of the triple

class HASH_TABLE1  export

    startup,do_adding,do_deleting,do_retrieving

inherit

    BASIC_TABLE

feature

 startup is
  -- Initialize the blocks at the start begining of the database
```

```
do
  initialize("Tree.env","Table1.dat","table1","free1");
end;

do_adding(text1,text2,text3:STRING) is
  -- Store the triple in the table

do
  add_triple(1,"Tree.env","table1","free1","Table1.dat",
             text1,text1,text2,text3);
end;

do_deleting(text1,text2,text3:STRING) is
  -- Remove the triple from the table

do
  delete_triple(1,"Tree.env","table1","free1","Table1.dat",
                text1,text1,text2,text3);
end;

do_retrieving(text1,text2,text3:STRING) is
  -- Retrieve the triples from the table

do
  retrieve_triple(1,"Tree.env","table1","Table1.dat",
                  text1,text1,text2,text3);
end;

end;
```

§ A.4 Class HASH_TABLE2

```
    -- The hash table that maps on the second part of the triple

class HASH_TABLE2  export

    startup,do_adding,do_deleting,do_retrieving

inherit

    BASIC_TABLE

feature

 startup is
  -- Initialize the blocks at the start begining of the database

  do
    initialize("Tree.env","Table2.dat","table2","free2");
  end;

 do_adding(text1,text2,text3:STRING) is
   -- Store the triple in the table

  do
    add_triple(2,"Tree.env","table2","free2","Table2.dat",
               text2,text1,text2,text3);
  end;

 do_deleting(text1,text2,text3:STRING) is
   -- Remove the triple from the table

  do
```

```
   delete_triple(2,"Tree.env","table2","free2","Table2.dat",
                 text2,text1,text2,text3);
 end;

do_retrieving(text1,text2,text3:STRING) is
 -- Retrieve the triples from the table

 do
   retrieve_triple(2,"Tree.env","table2","Table2.dat",
                   text2,text1,text2,text3);
 end;

end;
```

§ A.5 Class HASH_TABLE3

```
    -- The hash table that maps on the third part of the triple

class HASH_TABLE3  export

    startup,do_adding,do_deleting,do_retrieving

inherit

    BASIC_TABLE

feature

 startup is
  -- Initialize the blocks at the start begining of the database
```

```
 do
   initialize("Tree.env","Table3.dat","table3","free3");
 end;

do_adding(text1,text2,text3:STRING) is
  -- Store the triple in the table

 do
   add_triple(3,"Tree.env","table3","free3","Table3.dat",
              text3,text1,text2,text3);
 end;

do_deleting(text1,text2,text3:STRING) is
  -- Remove the triple from the table

 do
   delete_triple(3,"Tree.env","table3","free3","Table3.dat",
                 text3,text1,text2,text3);
 end;

do_retrieving(text1,text2,text3:STRING) is
 -- Retrieve the triples from the table

 do
   retrieve_triple(3,"Tree.env","table3","Table3.dat",
                   text3,text1,text2,text3);
 end;

end;
```

§ A.6 Class HASH_TABLE4

```
    -- The hash table that maps on the first and second part of the triple

class HASH_TABLE4  export

    startup,do_adding,do_deleting,do_retrieving

inherit

    BASIC_TABLE

feature

 startup is
  -- Initialize the blocks at the start begining of the database

  do
    initialize("Tree.env","Table4.dat","table4","free4");
  end;

 do_adding(text1,text2,text3:STRING) is
   -- Store the triple in the table

  local
    tmp_text: STRING;

  do
    tmp_text.Create(100);
    tmp_text.append(text1);
    tmp_text.append(text2);
    add_triple(4,"Tree.env","table4","free4","Table4.dat",
```

```
                tmp_text,text1,text2,text3);
 end;

do_deleting(text1,text2,text3:STRING) is
  -- Remove the triple from the table

 local
   tmp_text: STRING;

 do
   tmp_text.Create(100);
   tmp_text.append(text1);
   tmp_text.append(text2);
   delete_triple(4,"Tree.env","table4","free4","Table4.dat",
                 tmp_text,text1,text2,text3);
 end;

do_retrieving(text1,text2,text3:STRING) is
 -- Retrieve the triples from the table

 local
   tmp_text: STRING;

 do
   tmp_text.Create(100);
   tmp_text.append(text1);
   tmp_text.append(text2);
   retrieve_triple(4,"Tree.env","table4","Table4.dat",
                   tmp_text,text1,text2,text3);
 end;

end;
```

§ A.7 Class HASH_TABLE5

```
    -- The hash table that maps on the second and third part of the triple

class HASH_TABLE5  export

    startup,do_adding,do_deleting,do_retrieving

inherit

    BASIC_TABLE

feature

 startup is
  -- Initialize the blocks at the start begining of the database

  do
    initialize("Tree.env","Table5.dat","table5","free5");
  end;

 do_adding(text1,text2,text3:STRING) is
   -- Store the triple in the table

  local
    tmp_text: STRING;

  do
    tmp_text.Create(100);
    tmp_text.append(text2);
    tmp_text append(text3);
    add_triple(5,"Tree.env","table5","free5","Table5.dat",
               tmp_text,text1,text2,text3);
```

```
 end;

do_deleting(text1,text2,text3:STRING) is
  -- Remove the triple from the table

 local
   tmp_text: STRING;

 do
   tmp_text.Create(100);
   tmp_text.append(text2);
   tmp_text.append(text3);
   delete_triple(5,"Tree.env","table5","free5","Table5.dat",
                 tmp_text,text1,text2,text3);
 end;

do_retrieving(text1,text2,text3:STRING) is
 -- Retrieve the triples from the table

 local
   tmp_text: STRING;

 do
   tmp_text.Create(100);
   tmp_text.append(text2);
   tmp_text.append(text3);
   retrieve_triple(5,"Tree.env","table5","Table5.dat",
                   tmp_text,text1,text2,text3);
 end;

end;
```

§ A.8 Class HASH_TABLE6

```
    -- The hash table that maps on the first and third part of the triple

class HASH_TABLE6  export

    startup,do_adding,do_deleting,do_retrieving

inherit

    BASIC_TABLE

feature

 startup is
  -- Initialize the blocks at the start begining of the database

  do
    initialize("Tree.env","Table6.dat","table6","free6");
  end;

 do_adding(text1,text2,text3:STRING) is
  -- Store the triple in the table

  local
    tmp_text: STRING;

  do
    tmp_text.Create(100);
    tmp_text.append(text1);
    tmp_text.append(text3);
    add_triple(6,"Tree.env","table6","free6","Table6.dat",
               tmp_text,text1,text2,text3);
```

```
 end;

do_deleting(text1,text2,text3:STRING) is
  -- Remove the triple from the table

 local
   tmp_text: STRING;

 do
   tmp_text.Create(100);
   tmp_text.append(text1);
   tmp_text.append(text3);
   delete_triple(6,"Tree.env","table6","free6","Table6.dat",
                 tmp_text,text1,text2,text3);
 end;

do_retrieving(text1,text2,text3:STRING) is
 -- Retrieve the triples from the table

 local
   tmp_text: STRING;

 do
   tmp_text.Create(100);
   tmp_text.append(text1);
   tmp_text.append(text3);
   retrieve_triple(6,"Tree.env","table6","Table6.dat",
                   tmp_text,text1,text2,text3);
  end;

end;
```

§ A.9 Class HASH_TABLE7

```
    -- The hash table that maps on the first, second, and, third part of the
    -- triple

class HASH_TABLE7  export

   startup,do_adding,do_deleting,do_retrieving

inherit

   BASIC_TABLE

feature

 startup is
  -- Initialize the blocks at the start begining of the database

  do
    initialize("Tree.env","Table7.dat","table7","free7");
  end;

 do_adding(text1,text2,text3:STRING) is
   -- Store the triple in the table

  local
    tmp_text: STRING;

  do
    tmp_text.Create(150);
    tmp_text.append(text1);
    tmp_text.append(text2);
    tmp_text.append(text3);
```

```
    add_triple(7,"Tree.env","table7","free7","Table7.dat",
                tmp_text,text1,text2,text3);
  end;

do_deleting(text1,text2,text3:STRING) is
  -- Remove the triple from the table

  local
    tmp_text: STRING;

  do
    tmp_text.Create(150);
    tmp_text.append(text1);
    tmp_text.append(text2);
    tmp_text.append(text3);

    delete_triple(7,"Tree.env","table7","free7","Table7.dat",
                  tmp_text,text1,text2,text3);
  end;

do_retrieving(text1,text2,text3:STRING) is
  -- Retrieve the triples from the table

  local
    tmp_text: STRING;

  do
    tmp_text.Create(150);
    tmp_text.append(text1);
    tmp_text.append(text2);
    tmp_text.append(text3);

    retrieve_triple(7,"Tree.env","table7","Table7.dat",
```

```
                    tmp_text,text1,text2,text3);
  end;

end;
```

BIBLIOGRAPHY

[1] *An Introduction to Programming in C.* Al Kelley and Ira Pohl, 1984.

[2] *Eiffel: The Environment.* Interactive Software Engineering Inc., Oct. 1990.

[3] *Eiffel: The Libraries.* Interactive Software Engineering Inc., Oct. 1990.

[4] J. R. Abrial. Data Semantics. *In Data Base Management — Proceedings IFIP Working Conference on Data Base Management, edited J. W. Klimbie and K.L. Koffeman*, pages 1–59, North-Holland Publishing Company, 1974.

[5] H.M. Al-Haddad, K.M. George, and M.H. Samadzadeh. Approches to reusuability in C++ and Eiffel. *Journal of Object-Oriented Programming, Vol. 4 Issue: 5*, pages 34–35, Sept. 1991.

[6] M. Azmoodeh. BRMQ: A Database Interface Facility based on Graph Traversals and Extended Relationships on Group of Entities. *The Computer Journal, Vol. 33, No. 1*, pages 31–39, 1990.

[7] E. Balagurusamy. *Programming in C.* Tata Mc-Graw Hill Publishing Company, New Delhi, 1991.

[8] R.G.G. Cattell. *The Object Database Standard.* ODMG-93, Morgan Kaufmann Publishers, San Mateo, California, 1993.

[9] J. Cheng. A reusabilitry based software development environment. *ACM SIGSOFT, Software Engineering Notes Vol. 19 No. 2,*, pages 57–62, April 1994.

[10] B. Czejdo and M.C. Taylor. Integration of Database Sytems and Smalltalk. *IEEE Symposium on Applied Computing*, pages 393–402, 1991.

[11] B. D. Czejdo and M. C. Taylor. Integration of object-oriented programming languages and database systems in KOPERNIK. *Data & Knowledge Engineering*, pages 271–298, 1992.

[12] Beshir E.M.A. Elgalal. Minimally-redundant data structures and reasonable hyopthesis: some general heuristic methods of knowledge processing. *Ph.D. thesis, University of Stratclyde*, 1985.

[13] G. Birtwistle et al. Simula Begin. *Berlin, Germany: Studentliteratur and Auerbach*, 1973.

[14] R.A. Frost. Algorithms supplement. *The Computer Journal, Vol. 24, No. 4*, pages 383–384, 1981.

[15] R.A. Frost. ASDAS — A Simple Database Management System. *Proceedings of the 6th ACM European Regional Conference on Systems Architecture*, pages 232–240, 1981.

[16] R.A. Frost. Binary-Relational Storage Structures. *The Computer Journal, Vol 25, No. 3*, pages 358–367, 1982.

[17] R.A. Frost. *Introduction to Knowledge Base Systems*. Collins Professional and Technical Books, London, 1987.

[18] R.A. Frost and M.M. Peterson. A Function for Generating Nearly Balanced Binary Search Trees from Sets of Non-random Keys. *Software — Practice ane Experience, Vol. 12*, pages 163–168, 1982.

[19] D. McLeod H. Afsarmanesh, D. Knapp and A. Parker. An Object-Oriented Approach to Extensible Databases for VLSI/CAD. *Proceedings of the XI International Conference on Very Large Databases, Stockholm*, September, 1985.

[20] Ellis Horowitz and Sartaj Sahani. *Fundamentals of Data Structures*. Computer Science Press Inc., 1983.

[21] R. Howard. The Eiffel Programming Language. *Dr. Dobb's Journal, Vol. 18 Issue: 11*, pages 68–73, Oct. 1993.

[22] A. R. Hurson and Simin H. Pakzad. Object-Oriented Database Management Systems: Evolution and Performance Issues. *The Computer Journal,*, February, 1993.

[23] Won Kim. Object-Oriented Databases: Definition and Research Directions. *IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 3*, September 1990.

[24] Stephen Kochan and Patrick Wood. *Topics in C Programming*. Hayden Books Unix Library, 1989.

[25] P. Larson. Dynamic Hashing. *BIT, Vol. 18*, pages 184–201, 1978.

[26] Liwu Li. Lecture Notes in Object-oriented Databases. *Course 60–537, University of Windsor*, 1993.

[27] J.A. Mariani. Oggetto: An Object-Oriented Database Layered on a Triple Store. *The Computer Journal, Vol 35, No. 2*, pages 108–118, 1992.

[28] J.A. Mariani and Robert Lougher. TripleSpace: an experiment in a 3D graphical interface to a binary relational database. *Interacting with Computers Vol. 4 No. 2*, pages 147–162, 1992.

[29] D.R. McGregor and J.R. Malone. The FACT database: A system based on inferential methods. *Proceedings of the Cambridge Symposium Reasearch and Development in Information Retrieval*, June 1980.

[30] Bertrand Meyer. *Object-oriented Software Construction*. Prenctice Hall International Series in Computer Science, 1988.

[31] Bertrand Meyer. *Eiffel: The Language*. Prenctice Hall International Series in Computer Science, 1992.

[32] J. Micallef. Encapsulation, reusablilty, and extensibility in object-oriented programming languages. *Object-Oriented Programming, vol. 1, no.1*, pages 12–36, April/May 1988.

[33] G.T. Nguyen and D. Rieu. Schema Evolution in object-oriented database systems. *Data and Knowledge Engineering (4)*, pages 43–47, 1989.

[34] A. Poulovassilis. The Implementation of FDL, a Functional Database Language. *The Computer Journal Vol. 35 No.2*, pages 119–128, 1992.

[35] A.L. Rector and S. Kay. Descriptive Models for Medical Records and Data Interchange. *6th Conference on Medical Informatics*, pages 230–234, 1989.

[36] Marc J. Rochkind. *Advanced Unix Programming*. Prentice-Hall Software Series, 1985.

[37] R. Rock-Evans. Data Analysis. *Computer Weekly, IPC Electronic Press Ltd.*, 1980.

[38] M. E. Senko. The DDL in the context of a multilevel structured description: in DIAM II with FLORAL. *In Data Base Description, edited by B. C. M. Douque and G. M. Nijssen, North-Holland Publishing Company*, pages 239–258, 1975.

[39] M.E. Senko, E.B. Altman, M.M. Astrahan, and P.L. Fehder. Data Structures and accessing in database systems. *IBM Systems Journal, Vol. 12, No. 1*, pages 30–93, 1973.

[40] G.O.H. Sharman and N. Winterbottom. NBD: Non-Programmer Database Facility. *IBM Technical Report TR 12.179*, 1979.

[41] M.J.R. Shave. Entities, functions and binary relations: steps to a conceptual scheme. *The Computer Journal 24 (No. 1)*, pages 42–45, 1981.

[42] Rajesh Shenoy. Object-Oriented Database Management Systems : A survey. *60–510 Background Reading, School of Computer Science, University of Windsor*, 1993.

[43] C. Snape. OSROS: A Binary Relational Database System . *Bsc Project Internal Report, Computing Department, Lancaster University*, 1986.

[44] W. R. Stevens. *Unix Network Programming*. Prentice-Hall of India Pvt. Ltd., New Delhi, 1992.

VITA AUCTORIS

Rajesh Shenoy was born in 1968 in **India**. He graduated from **Parle College, Bombay, India** in 1986. From there he went on to the **University of Poona, Poona, India** where he obtained a Bachelor of Engineering (B. E.) in Computer Science in 1990. He is currently a candidate for the Master's degree in Computer Science at the University of Windsor and hopes to graduate in the Fall of 1994.